

Intro

Here we are again. I got a bit of a taste for this after the Lemmings tutorial a short time ago so I jumped straight back in again. This time I've chosen Shadow of the Beast (CAPS/SPS release 1357). This is a classic Amiga game that pretty much everyone will be familiar with. The gameplay isn't so great – it's a tough game to beat – but it still looks and sounds pretty awesome even today.

This one is roughly on a par with Lemmings in terms of difficulty so again not necessarily the best tutorial for a beginner. As with the previous tutorial I'll be using WinUAE configured to emulate an A500 with the following:

- Action Replay III Cartridge – useful for general poking around, reading disk data into RAM and messing with it. Good for assembling very small bits of code to test. (Sometimes referred to as AR from now on).
- 2MB Fast RAM highly useful for dealing with large disk image data in memory with Action Replay. Located at \$200000 - \$400000 on my setup.

In addition the following software will be used:

RNC Propack – used to pack the files

XFDMaster – used to unpack the files

Devpac – very briefly used to create an exe (you can use pretty much any decent assembler)

Investigation

First thing we need to do is figure out what kind of protections we might need to handle. So here's the obligatory screenshot of the copy program that shows that we are dealing with a custom MFM disk (see figure 1a). So as usual we start from the bootblock (see figure 1b – 1f). There's a little bit of setup code (\$40000-\$40050) then what looks like a call to a depack routine to depack the rest of the bootblock code to \$8. Then it starts executing from address \$18. The depack code looks like a familiar bytekiller routine. I don't like the look of this depacking to address \$8 so I'm going to force it to unpack to a different address so that it doesn't interfere with Action replay. See figure 1g – we will unpack to \$60000.

Now we can have a look at the resulting depacked code (see figure 2a-2k). We see some weird checksum type stuff at \$60010 to \$6003c that seems to serve no useful purpose and we can simply skip it. It doesn't produce any kind of output that could be checked later so we can ignore it. The more interesting stuff starts around \$60072. We can see a couple of calls to a subroutine and then a jump to \$68058. Bearing in mind that our disassembly at \$60000

should have been loaded in low memory starting at address \$8 so we don't currently have any knowledge of what's at \$68058 so the two routines must do something interesting.

If we take the second call starting at \$60268 in our disassembly we see the same unpack routine we saw in the bootblock. We can see that the call to that routine sets a value in A1 before the call, so it's a good bet that the parameters are source in A0, and destination in A1. This makes sense as it will then be unpacking to \$68000 which ties in with the JMP \$68058 that follows. If we now take a look at the first subroutine we see various instructions related to disk activity so it's probably a load routine. The register setup stuff around \$60186 to \$60220 is a dead giveaway. The presence of the \$55555555 is also a strong indicator that it's probably doing something interesting.

Looking through the loader code it's a very compact and quite basic routine. First of all it selects the drive and then it reads the longword at address in register A1. It then divides the value by \$1838. This looks like a track length so I'm thinking that the value in (a1) is the byte offset to start reading from. Then there's some stuff that seems to prevent the value being too high and adjusting it down if it is. Not sure what this is, maybe just some kind of checking to ensure it doesn't try to step the heads too far.

The routine at \$6011c seems to be a routine to locate a particular track but it only seems capable of stepping in one direction and it doesn't check or calculate the location of the heads before it starts. It is simply assuming that the heads will be at track 0 after executing the bootblock. After this we read another longword from (a1) and perform a shift and a subtraction. This looks like a conversion from bytes into longs and a subtraction to use the value in a DBF loop. This would appear to be the length of the data to read. We then go into a loop executing two subroutines one after the other. The first appears to load a track's worth of data and then the second steps the heads ready for the next track. If we look at the values that will be passed at the address in A1 we see the values in figure 21. The disk start offset is \$190c and the length is \$b40

It's interesting to note that the side of the disk being read is never changed so it simply steps the heads after every track and keeps reading from the same side. This routine isn't capable of reading from both sides of the disk. If we want to use this routine to extract all of the data from the two disks we will need to modify it slightly as we go.

It also seems that the code executes an infinite loop, however if you look at \$60238-\$6023e when the condition is met it adds 4 to A7 which removes the return address from the stack causing it to exit the whole load routine when the RTS is executed at \$6025e. This is the way it drops out of the infinite loop.



Figure 1a – screenshot of MFM protected disk.

```

rt 0 1 40000
Disk ok
d 4000c
~04000C LEA 40020(PC),A0
~040010 MOVE.L A0,00000020.S
~040014 NOP
~040016 NOP
~040018 MOVE.W #2700,SR
~04001C NOP
~04001E NOP
~040020 MOVE.W #2700,SR
~040024 MOVE.W #7FFF,00DFF096
~04002C MOVE.W #8210,00DFF096
~040034 MOVE.W #7FFF,00DFF09A
~04003C MOVE.W #7FFF,00DFF09C
~040044 MOVEA.L 00000010.S,A1
~040048 LEA 4005C(PC),A0
~04004C MOVE.L A0,00000010.S
~040050 LINEF

```

Figure 1b – bootblock(1)

```

d 4005c
~04005C MOVE.L A1,00000010.S
~040060 LEA 40126(PC),A0
~040064 LEA 00000000.S,A1
~040068 BSR 00040070
~04006C JMP 00000010.S
=====
^040070 ADDA.L (A0),A0
~040072 MOVEA.L -(A0),A2
~040074 ADDA.L A1,A2
~040076 MOVE.L -(A0),D5
~040078 MOVE.L -(A0),D0
~04007A EOR.L D0,D5
~04007C LSR.L #1,D0
~04007E BNE 00040082
~040080 BSR 000400F4
~040082 BCS 000400B6
~040084 MOVEQ #8,D1
~040086 MOVEQ #1,D3
~040088 LSR.L #1,D0
~04008A BNE 0004008E
~04008C BSR 000400F4
~04008E BCS 000400DC
~040090 MOVEQ #3,D1

```

Figure 1c – bootblock(2)

```

~040092 CLR.W D4
~040094 BSR 00040100
~040096 MOVE.W D2,D3
~040098 ADD.W D4,D3
~04009A MOVEQ #7,D1
~04009C LSR.L #1,D0
~04009E BNE 000400A2
~0400A0 BSR 000400F4
~0400A2 RORL.L #1,D2
~0400A4 DBF D1,0004009C
~0400A8 MOVE.B D2,-(A2)
~0400AA DBF D3,0004009A
~0400AE BRA 000400E8
=====
^0400B0 MOVEQ #8,D1
~0400B2 MOVEQ #8,D4
~0400B4 BRA 00040094
=====
^0400B6 MOVEQ #2,D1
~0400B8 BSR 00040100
~0400BA CMP.B #2,D2
~0400BE BLT 000400D2
~0400C0 CMP.B #3,D2
~0400C4 BEQ 000400B0
~0400C6 MOVEQ #8,D1

```

Figure 1d – bootblock(3)

```

~0400CA MOVE.W D2,D3
~0400CC MOVE.W #C,D1
~0400D0 BRA 000400DC
=====
^0400D2 MOVE.W #9,D1
~0400D6 ADD.W D2,D1
~0400D8 ADDQ.W #2,D2
~0400DA MOVE.W D2,D3
~0400DC BSR 00040100
~0400DE SUBQ.W #1,A2
~0400E0 MOVE.B 0(A2,D2.W),(A2)
~0400E4 DBF D3,000400DE
~0400E8 CMPA.L A2,A1
~0400EA BLT 0004007C
~0400EE TST.L D5
~0400F0 BNE 0004011A
~0400F2 BRA 00040122
=====
^0400F4 MOVE.L -(A0),D0
~0400F6 EOR.L D0,D5
~0400F8 MOVE.B #10,CCR
~0400FC ROXR.L #1,D0
~0400FE RTS
=====
^040100 SUBQ.W #1,D1

```

Figure 1e – bootblock(4)

```

~040102 CLR.W D2
~040104 LSR.L #1,D0
~040106 BNE 00040112
~040108 MOVE.L -(A0),D0
~04010A EOR.L D0,D5
~04010C MOVE.B #10,CCR
~040110 ROXR.L #1,D0
~040112 ROXL.L #1,D2
~040114 DBF D1,00040104
~040118 RTS
=====
^04011A MOVE.L #FFFFFFF,D0
~040120 RTS
=====
^040122 MOVEQ #0,D0
~040124 RTS
=====
^040126 ADD.W #04,D0

```

Figure 1f – bootblock(5)

```

a 50000
^050000 lea 40126,a0
^050006 lea 60000,a1
^05000C jsr 40070
^050012 bra 50012
^050014
bs 50012
Breakpoint inserted
Ready.
g 50000
No known virus in memory!
Ready.
Breakpoint raised at address: 00050012
-

```

Figure 1g – call depack routine to a sensible location

```

d 60000
~060000 ORI.B #4C,D5
~060004 ORI.B #70,D5
~060008 ORI.B #C,D0
~06000C ORI.B #40,D0
~060010 MOVEQ #0,D1
~060012 MOVE.L 00000008,S,D0
~060016 LEA 00000044,A0
~06001C MOVE.L (A0)+,D2
~06001E EOR.L D2,D0
~060020 MULU.W #127D,D0
~060024 SUBQ.L #7,D0
~060026 ADD.L D0,D1
~060028 CMPA.L #326,A0
~06002E BLT 0006001C
~060030 SUB.L 0000000C,S,D1
~060034 CMP.L #A8E85B3,D1
~06003A BNE 0006001C
~06003C CLR.W 00DFF180
~060042 BSET #1,00BFE001
~06004A MOVEA.L #400,A7
~060050 BRA 00060072
;=====
^060052 ORI.B #1,D0
_060056 ORI.B #3,D2

```

Figure 2a – depacked code (1)

```

~06005A ORI.B #5,D4
~06005E ORI.B #7,D6
~060062 ORI.B #9,A0
~060066 ORI.B #B,A2
~06006A ORI.B #D,A4
~06006E ORI.B #F,A6
~060072 BSR 00060094
~060076 LEA 00067000,A0
~06007C LEA 00000010,S,A1
~060080 BSR 00060144
~060084 LEA 00068000,A1
~06008A BSR 00060268
~06008E JMP 00068058
;=====
^060094 MOVE.W #10,00DFF096
~06009C MOVE.W #2,00DFF09C
~0600A4 MOVE.W #7FFF,00DFF09E
~0600AC MOVE.W #8100,00DFF09E
~0600B4 ORI.B #78,00BFD100
~0600BC BCLR #7,00BFD100
~0600C4 BCLR #3,00BFD100
~0600CC MOVE.W #BB8,D6
~0600D0 DBF D6,000600D0
~0600D4 BTST #5,00BFE001
_0600DC BNE 000600D4

```

Figure 2b – depacked code (2)

```

~0600DE LEA    60262(PC),A3
~0600E2 MOVE.W #1,(A3)
~0600E6 RTS
;=====
^0600E8 BCLR   #1,00BFD100
~0600F0 BCLR   #0,00BFD100
~0600F8 BSET   #0,00BFD100
~060100 BSR    00060136
~060104 MOVE.L A0,-(A7)
~060106 LEA    60260(PC),A0
~06010A ADDQ.W #1,(A0)
~06010C MOVEA.L (A7)+,A0
~06010E BRA    0006012A
;=====
^060112 CMP.W  60260(PC),D0
~060116 BEQ    0006011A
~060118 BGT    0006011C
~06011A RTS
;=====
^06011C SUB.W  60260(PC),D0
~060120 SUBQ.W #1,D0
~060122 BSR    000600E8
~060124 DBF    D0,00060122
~060128 RTS
;=====

```

Figure 2c – depacked code (3)

```

~06012A BTST   #5,00BFE001
~060132 BNE    0006012A
~060134 RTS
;=====
^060136 MOVE.W D7,-(A7)
~060138 MOVE.W #1388,D7
~06013C DBF    D7,0006013C
~060140 MOVE.W (A7)+,D7
~060142 RTS
;=====
^060144 MOVE.L A0,-(A7)
~060146 BSR    0006014E
~06014A MOVEA.L (A7)+,A0
~06014C RTS
;=====
^06014E BSR    00060094
~060152 MOVE.L (A1)+,D0
~060154 MOVE.L D0,D3
~060156 DIVU.W #1838,D0
~06015A CMP.W  #56,D0
~06015E BLT    0006016A
~060160 SUBI.L #822D0,D3
~060166 SUBI.W #56,D0
~06016A MOVEQ  #0,D2
~06016C MOVE.W D0,D2

```

Figure 2d – depacked code (4)

```

~06016E BSR      00060112
~060170 MULL.W  #1838,D2
~060174 SUB.L   D2,D3
~060176 MOVE.L  (A1),D4
~060178 LSR.L   #2,D4
~06017A SUBQ.L  #1,D4
~06017C BSR      00060186
~060180 BSR      000600E8
~060184 BRA      0006017C
=====
λ 060186 MOVE.W  #F,D2
~06018A MOVE.L  #1860,D0
~060190 MOVEA.L 60264(PC),A6
~060194 MOVE.W  #2,00DFF09C
~06019C MOVE.L  A6,00DFF020
~0601A2 MOVE.W  #8210,00DFF096
~0601AA MOVE.W  #4489,00DFF07E
~0601B2 MOVE.W  #7F00,00DFF09E
~0601BA MOVE.W  #B500,00DFF09E
~0601C2 MOVE.W  #4000,00DFF024
~0601CA MOVE.B  00BFD000,D1
~0601D0 MOVE.B  00BFD000,D1
~0601D6 BTST    #4,D1
~0601DA BEQ      000601D0
~0601DC ADDI.W  #8001,D0

```

Figure 2e – depacked code (5)

```

~0601E0 MOVE.W  D0,00DFF024
~0601E6 MOVE.W  D0,00DFF024
~0601EC MOVE.W  00DFF01E,D0
~0601F2 BTST    #1,D0
~0601F6 BEQ      000601EC
~0601F8 MOVE.L  (A6)+,D0
~0601FA MOVE.L  (A6)+,D1
~0601FC ASL.L   #1,D0
~0601FE ANDI.L  #AAAAAAA,D0
~060204 ANDI.L  #55555555,D1
~06020A OR.L    D1,D0
~06020C CMP.L   #534F5442,D0
~060212 BEQ      00060222
~060214 DBF     D2,0006018A
~060218 MOVE.W  #F00,00DFF180
~060220 BRA      00060218
=====
λ 060222 MOVEQ   #0,D2
~060224 MOVE.W  #60D,D2
~060228 TST.W   D3
~06022A BEQ      00060236
~06022C ADD.W   D3,D3
~06022E ADDA.L  D3,A6
~060230 LSR.W   #3,D3
~060232 SUB.W   D3,D2

```

Figure 2f – depacked code (6)

```

~060234 CLR.W    D3
~060236 CMP.L    D2,D4
~060238 BGT      00060240
~06023A MOVE.W   D4,D2
~06023C ADDQ.L   #4,A7
~06023E BRA      00060244
=====
^060240 SUB.L     D2,D4
~060242 SUBQ.L   #1,D4
~060244 MOVE.L   (A6)+,D0
~060246 MOVE.L   (A6)+,D1
~060248 ADD.L     D0,D0
~06024A ANDI.L   #AAAAAAA,D0
~060250 ANDI.L   #5555555,D1
~060256 OR.L     D1,D0
~060258 MOVE.L   D0,(A0)+
~06025A DBF      D2,00060244
~06025E RTS
=====
^060260 ORI.B    #0,D0
~060264 ORI.B    #0,D0
~060268 ADDA.L   (A0),A0
~06026A MOVEA.L  -(A0),A2
~06026C ADDA.L   A1,A2
~06026E MOVE.L   -(A0),D5

```

Figure 2g – depacked code (7)

```

~060270 MOVE.L   -(A0),D0
~060272 EOR.L    D0,D5
~060274 LSR.L    #1,D0
~060276 BNE      0006027A
~060278 BSR      000602EC
~06027A BCS      000602AE
~06027C MOVEQ    #8,D1
~06027E MOVEQ    #1,D3
~060280 LSR.L    #1,D0
~060282 BNE      00060286
~060284 BSR      000602EC
~060286 BCS      000602D4
~060288 MOVEQ    #3,D1
~06028A CLR.W    D4
~06028C BSR      000602F8
~06028E MOVE.W   D2,D3
~060290 ADD.W    D4,D3
~060292 MOVEQ    #7,D1
~060294 LSR.L    #1,D0
~060296 BNE      0006029A
~060298 BSR      000602EC
~06029A RORL.L   #1,D2
~06029C DBF      D1,00060294
~0602A0 MOVE.B   D2,-(A2)
~0602A2 DBF      D3,00060292

```

Figure 2h – depacked code (8)

```

~0602A6 BRA      000602E0
;=====
^0602A8 MOVEQ    #8,D1
~0602AA MOVEQ    #8,D4
~0602AC BRA      0006028C
;=====
^0602AE MOVEQ    #2,D1
~0602B0 BSR      000602F8
~0602B2 CMP.B    #2,D2
~0602B6 BLT      000602CA
~0602B8 CMP.B    #3,D2
~0602BC BEQ      000602A8
~0602BE MOVEQ    #8,D1
~0602C0 BSR      000602F8
~0602C2 MOVE.W   D2,D3
~0602C4 MOVE.W   #C,D1
~0602C8 BRA      000602D4
;=====
^0602CA MOVE.W   #9,D1
~0602CE ADD.W    D2,D1
~0602D0 ADDQ.W   #2,D2
~0602D2 MOVE.W   D2,D3
~0602D4 BSR      000602F8
~0602D6 SUBQ.W   #1,A2
~0602D8 MOVE.B   0(A2,D2.W),(A2)

```

Figure 2i – depacked code (9)

```

~0602DC DBF      D3,000602D6
~0602E0 CMPA.L   A2,A1
~0602E2 BLT      00060274
~0602E6 TST.L    D5
~0602E8 BNE      00060312
~0602EA BRA      0006031A
;=====
^0602EC MOVE.L   -(A0),D0
~0602EE EOR.L    D0,D5
~0602F0 MOVE.B   #10,CCR
~0602F4 ROXR.L   #1,D0
~0602F6 RTS
;=====
^0602F8 SUBQ.W   #1,D1
~0602FA CLR.W    D2
~0602FC LSR.L    #1,D0
~0602FE BNE      0006030A
~060300 MOVE.L   -(A0),D0
~060302 EOR.L    D0,D5
~060304 MOVE.B   #10,CCR
~060308 ROXR.L   #1,D0
~06030A ROXL.L   #1,D2
~06030C DBF      D1,000602FC
~060310 RTS
;=====

```

Figure 2j – depacked code (10)

```

;=====
^060312 MOVE.L   #FFFFFFF,D0
~060318 RTS
;=====
^06031A MOVEQ    #0,D0
~06031C RTS
;=====
^06031E ORI.B    #0,D0

```

Figure 2k – depacked code (11)

```
n 60008  
:060008 00 00 19 0C 00 00 0B 40 72 00 20 38 00 08 41 F9 .....Dr. 8..A.
```

Figure 2l – disk offset and length (\$190c and \$b40)

Extracting the MFM data

If the tracks are \$1838 in length and the routine only handles one side of the disk we can probably extract the whole of one side in one go and repeat the process for both sides of the disk and hopefully for both disks. We will need to store the resulting file storing the data for each side of the original disk onto a separate blank disk requiring 4 disks in total.

Also we need to consider that the loader does not start off by seeking track 0, so it assumes that the drive will be at track 0 after loading the bootblock. A little trick I learnt to deal with these kinds of loaders is to reboot and insert a non-bootable disk into the drive before dropping into AR. When it detects the non-bootable disk it will attempt to read the bootblock and leave the heads at track 0. Once the hand appears you can then drop into AR and swap disks and use the RT command to read in the bootblock and you are ready to go. Depack the code to \$60000 again and then we write a little bit of code (see figure 3a) and run it. The code shown starts at offset \$1838 (so skips track 0). Once this is complete we can save off the result to the first of 4 disks.

Next we can try the second disk and see if we can extract the contents of the whole of side 0 starting at track 0. Reboot and repeat the same steps we did for disk 1 with the non-bootable disk trick, then load and depack bootblock and set up code at \$50000 to load. Insert disk 2 and run the code (see figure 3b).

Next if we want to extract the other side of the two disks we need to change the loader slightly to select the other side of the disk. First reboot again and repeat the steps as previously then make the changes as per figure 3c. The result of this will be that it sticks on track 0 and never completes. If we jump into AR we see we are stuck in a loop waiting for the sync marker (see figure 3d). This suggests that either there is a different format for side 1 or that track 0 on side 1 contains something other than the format used on the rest of the tracks (maybe a RNC Copylock or similar). We can simply try skipping track 0 and see how we get on. Try again using the changes shown in figure 3e and now it completes and we can save off the data.

Now repeat the same steps for disk 2 side 1. Again we see that track 0 causes us problems so we can assume some kind of security track on both disks. Skip track 0 and extract tracks 1-79. (See figure 3f).

Now that we have all of the raw MFM data we can start to debug further through the load sequence. Repeat the same actions as before until we get the code depacked to \$60000 again. Take a look at the code in figure 3g. The data at \$60008 is the data that would be at \$10 if we were following the correct boot sequence. We can build a similar piece of code that does not assume we are loaded into low memory (see figure 3h). Now using the memory

dump command we can see that some data has been loaded to \$67000 and depacked to \$68000. Keep scrolling and you will see it ends around the \$69240 mark (see figure 3i).

Save this data off to a disk using the SM command to save memory area \$68000 to \$69f240 (this is included in the archive along with this tutorial called BEAST_LOAD. If we now try and execute the loader from \$68058 as we saw it do in the previous code we find that it doesn't work too well. The loader expects to be called in supervisor mode so if we knock up a quick piece of code to handle that we can run the loader (see figure 3j). Don't forget to insert the original disk 1 before executing the G 60000. The game will load fully if you start the game you will notice that the game crashes quite soon after you start. We will look into this later.

Reboot and load up the loader to \$68000 and redo the supervisor code at \$60000 as in figure 3j. Try running it again and you will see this time it gets stuck. Let's take a look at the loader code and see if we can see what's going on. Search for access to DFF020 (disk pointer register) to try and find the load routine (see figure 3k). Scroll back and you will find something at \$690c4 that looks similar to the previous load routine we found (see figure 3l).

The main bulk of this code seems to match the previous loader but there are some small differences. Notice the extra code at \$690dc to \$690e8. This is not present in the original routine. Figure 3n shows the two routines. One or the other is called depending on whether the disk start offset is greater than \$822d0 or not. The routines both select drive 0 but they differ in the side of the disk that is selected. So the loader will load from side 0 or side 1 depending on whether the disk offset is above \$822d0 or not. It later subtracts \$822d0 from the disk offset if it needs to (we saw this seemingly redundant piece of code in the first loader we looked at too). So passing in the start offset of \$822d0 that would result in the loader reading from the start of side 1.

Also if we look at the track seek routine at \$69006 we see more differences. This loader is able to seek the heads in both directions based on the current track position which is stored in 691fc. There still seems to be no routine to seek to track 0 initially, so it's probable that this loader also assumes the track position after the previous load (as the previous load routine did too). If we take a look at \$691fc we see that the loader always expects the track position to be at track 1 when it is called for the first time. That explains why it failed to run properly after we rebooted.

Now if we take a look at the places where this routine is called from inside our code between \$68000 and \$69240 (see figure 4a and 4b). we see that the values passed in A1 in each case are between \$69200 and \$69238. This area seems to be a file table and we can see the contents of this area in figure 4c.

So we can see the following files:

Address	Disk start offset	Length
69200	244c	313d0
69208	3381c	2ecc
69210	366e8	48ec
69218	3afd4	6ce0
69220	41cb4	1f400
69228	610b4	f64
69230	62018	9e84
69238	6be9c	10a4

As we can see all of the start addresses are below \$822d0 so all of these files are loaded from side 0 of the disk. Using the raw data we extracted for side 0 of disk 1 we can potentially write a memory loader to see it loading from our raw data (see figure 4d). Using the same supervisor code from before and execute the loader and it will run up to the point where disk 2 is required with no disk access.

At this point pressing fire it attempts to load from the disk again. If you don't have one of the original disks inserted it will fail. If you insert disk 2 and press fire the game will continue to load again as it did previously. So reboot and repeat the process shown in figure 4d but this time jump into AR before pressing fire to continue loading.

```

a 50000
^050000 move.w #$7fff,$dff096
^050008 move.w #$7fff,$dff09a
^050010 lea $200000,a0
^050016 lea 270000,a1
^05001C move.l #1838,(a1)
^050022 move.l #77948,4(a1)
^05002A jsr 60144
^050030 bra 50030
^050032

bs 50030
Breakpoint inserted
Ready.
g 50000
No known virus in memory!
Ready.
Breakpoint raised at address: 00050030

sm beast1_0.raw, 280000 2f6110
Disk ok

```

Figure 3a – extract tracks 1-79 of disk 1 –side 0

```

a 50000
^050000 move.w #$7fff,$dff096
^050008 move.w #$7fff,$dff09a
^050010 lea 280000,a0
^050016 lea 270000,a1
^05001C move.l #0,(a1)
^050022 move.l #79180,4(a1)
^05002A jsr 60144
^050030 bra 50030
^050032

bs 50030
Breakpoint inserted
Ready.
g 50000
No known virus in memory!
Ready.
Breakpoint raised at address: 00050030

sm beast2_0.raw, 280000 2f9180
Disk ok

-

```

Figure 3b – extract tracks 0 – 79 of disk 2 side 0

```

~060094 MOVE.W #10,00DFF096
~06009C MOVE.W #2,00DFF09C
~0600A4 MOVE.W #7FFF,00DFF09E
~0600AC MOVE.W #8100,00DFF09E
~0600B4 ORI.B #78,00BFD100
~0600BC BCLR #7,00BFD100
~0600C4 BCLR #3,00BFD100
~0600CC MOVE.W #BB8,D6
~0600D0 DBF D6,000600D0
~0600D4 BTST #5,00BFE001
~0600DC BNE 000600D4
~0600DE LEA 60262(PC),A3
~0600E2 MOVE.W #1,(A3)
~0600E6 RTS

a 600c4
^0600C4 jsr 50100
^0600CA nop
^0600CC

a 50100
^050100 bclr #3,bfd100
^050108 bclr #2,bfd100
^050110 rts
^050112

-

```

Figure 3c – insert a patch into the loader to select side 1 instead of side 0.

```

P
D0=00003060 0000FF10 0000000F 00000000 0001E45F 00000000 FFFFFFFF 0000004F
A0=00280000 00270004 00060000 00060262 00001558 00C014B6 00002000 00C80000
PC = 000601F2 USP = 00C014A2 SR = 0000 T=0 S=0 I=000 X=0 N=0 Z=0 V=0 C=0
d 601c2
~0601C2 MOVE.W #4000,00DFF024
~0601CA MOVE.B 00BFDD00,D1
~0601D0 MOVE.B 00BFDD00,D1
~0601D6 BTST #4,D1
~0601DA BEQ 000601D0
~0601DC ADDI.W #8001,D0
~0601E0 MOVE.W D0,00DFF024
~0601E6 MOVE.W D0,00DFF024
~0601EC MOVE.W 00DFF01E,D0
~0601F2 BTST #1,D0
~0601F6 BEQ 000601EC
~0601F8 MOVE.L (A6)+,D0
~0601FA MOVE.L (A6)+,D1
~0601FC ASL.L #1,D0
~0601FE ANDI.L #AAAAAAA,D0

```

Figure 3d – stuck in a loop waiting for a sync marker

```

~050000 MOVE.W #7FFF,00DFF096
~050008 MOVE.W #7FFF,00DFF09A
~050010 LEA 00280000,A0
~050016 LEA 00270000,A1
~05001C MOVE.L #0,(A1)
~050022 MOVE.L #79180,4(A1)
~05002A JSR 00060144
~050030 BRA 00050030
=====
^050032 ORI.B #0,D0
~050036 ORI.B #0,D0
a 5001c
^05001C move.l #1838,(a1)
^050022 move.l #77948,4(a1)
^05002A
bs 50030
Breakpoint inserted
Ready.
g 50000
No known virus in memory!
Ready.
Breakpoint raised at address: 00050030
sm beast1_1.raw,280000 2f6110
Disk ok
-

```

Figure 3e – this time it worked

```

~0601FA MOVE.L (A6)+,D1
~0601FC ASL.L #1,D0
~0601FE ANDI.L #AAAAAAA,D0
~060204 ANDI.L #55555555,D1
~06020A OR.L D1,D0
~06020C CMP.L #534F5442,D0
d 601ec
~0601EC MOVE.W 00DFF01E,D0
~0601F2 BTST #1,D0
~0601F6 BEQ 000601EC
~0601F8 MOVE.L (A6)+,D0
a 5001c
^05001C move.l #1838,(a1)
^050022 move.l #77948,4(a1)
^05002A
g 50000
No known virus in memory!
Ready.
d
~050030 BRA 00050030
;=====
^050032 ORI.B #0,D0
sm beast_2_1.raw, 280000 2f6110
Disk ok
-

```

Figure 3f – disk 2 side 1 – tracks extracted

```

d 60072
~060072 BSR 00060094
~060076 LEA 00067000,A0
~06007C LEA 00000010,S,A1
~060080 BSR 00060144
~060084 LEA 00068000,A1
~06008A BSR 00060268
~06008E JMP 00068058
;=====
^060094 MOVE.W #10,00DFF096
~06009C MOVE.W #2,00DFF09C
~0600A4 MOVE.W #7FFF,00DFF09E
~0600AC MOVE.W #8100,00DFF09E
n 60008
:060008 00 00 19 0C 00 00 0B 40 72 00 20 38 00 00 41 F9 .....0r. 8..A.

```

Figure 3g – the next steps in the load sequence.

```

d 50000
~050000 MOVE.W #7FFF,00DFF096
~050008 MOVE.W #7FFF,00DFF09A
~050010 JSR 00060094
~050016 LEA 00067000,A0
~05001C LEA 00060008,A1
~050022 JSR 00060144
~050028 LEA 00068000,A1
~05002E JSR 00060268
~050034 BRA 00050034
;=====
^050036 ORI.B #0,D0
bs 50034
Breakpoint inserted
Ready.
g 50000
No known virus in memory!
Ready.
Breakpoint raised at address: 00050034
-

```

Figure 3h – our copied loader.

```

:069110 E4 8C 53 84 61 00 00 08 61 00 FE 98 60 F6 34 3C ä.S.a...a...'ö4<
:069120 00 0F 20 3C 00 00 18 60 2C 7A 00 CE 33 FC 00 02 ..<...z...3ü..
:069130 00 DF F0 9C 23 CE 00 DF F0 20 33 FC 82 10 00 DF ..β..#...β...3ü...β
:069140 F0 96 33 FC 44 89 00 DF F0 7E 33 FC 7F 00 00 DF ..3üD...β~3ü...β
:069150 F0 9E 33 FC B5 00 00 DF F0 9E 33 FC 40 00 00 DF ..3ü...β...3ü...β
:069160 F0 24 12 39 00 BF DD 00 12 39 00 BF DD 00 08 01 .$.9....9.....
:069170 00 04 67 F4 06 40 80 01 33 C0 00 DF F0 24 33 C0 ..g..@...3..β.$3.
:069180 00 DF F0 24 30 39 00 DF F0 1E 08 00 00 01 67 F4 .β.$09.β.....g.
:069190 20 1E 22 1E E3 80 02 80 AA AA AA AA 02 81 55 55 ..".....UU
:0691A0 55 55 80 81 B0 BC 53 4F 54 42 67 0E 51 CA FF 74 UU....SOTBg.Ö..t
:0691B0 33 FC 0F 00 00 DF F1 80 60 F6 74 00 34 3C 06 0D 3ü...β...'öt.4<..
:0691C0 4A 43 67 0A D6 43 DD C3 E6 4B 94 43 42 43 B8 82 JcG.ÖC...K.CBC..
:0691D0 6E 06 34 04 58 8F 60 04 98 82 53 84 20 1E 22 1E n.4.X...'S...'
:0691E0 E3 80 02 80 AA AA AA AA 02 81 55 55 55 55 80 81 .....UUUU..
:0691F0 20 C0 51 CA FF E8 4E 75 00 05 14 00 00 01 00 00 .Ö...Nu.....
:069200 00 00 24 4C 00 03 13 D0 00 03 38 1C 00 00 2E CC ..$L.....8.....
:069210 00 03 66 E8 00 00 48 EC 00 03 AF D4 00 00 6C E0 ..f...H.....l.
:069220 00 04 1C B4 00 01 F4 00 00 06 10 B4 00 00 0F 64 .....d
:069230 00 06 20 18 00 00 9E 84 00 06 BE 9C 00 00 10 A4 ..
:069240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:069250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:069260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:069270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Figure 3i – end of the depacked data.

```

d 60000
~060000 LEA    00068058,A0
~060006 MOVE.L A0,00000080
~06000C TRAP   #0

g 60000

```

Figure 3j – the code to run the loader in supervisor mode.

```

faq dff020 68000 69240
Search from: 068000 to: 069240
069134 MOVE.L   A6,00DFF020
Searched up to adr: 069240
Ready.
d 69134
~069134 MOVE.L   A6,00DFF020
~06913A MOVE.W   #8210,00DFF096
~069142 MOVE.W   #4489,00DFF07E
~06914A MOVE.W   #7F00,00DFF09E
~069152 MOVE.W   #B500,00DFF09E
~06915A MOVE.W   #4000,00DFF024
~069162 MOVE.B   00BFDD00,D1
~069168 MOVE.B   00BFDD00,D1
~06916E BTST     #4,D1
~069172 BEQ      00069168

```

Figure 3k – loader code

```

=====
λ 0690C4 MOVE.L A0,-(A7)
~0690C6 BSR 000690CE
~0690CA MOVEA.L (A7)+,A0
~0690CC RTS
=====
λ 0690CE LEA 691FE(PC),A3
~0690D2 TST.W (A3)
~0690D4 BNE 000690DA
~0690D6 BSR 00068F5E
~0690DA MOVE.L (A1)+,D0
~0690DC CMP.L #822D0,D0
~0690E2 BGE 000690E8
~0690E4 BSR 00069088
~0690E6 BRA 000690EA
=====
λ 0690E8 BSR 000690A6
~0690EA MOVE.L D0,D3
~0690EC DIVU.W #1838,D0
~0690F0 CMP.W #56,D0
~0690F4 BLT 00069100
~0690F6 SUBI.L #822D0,D3
~0690FC SUBI.W #56,D0
~069100 MOVEQ #0,D2

```

Figure 3l – start of loader code

```

~069102 MOVE.W D0,D2
~069104 BSR 00069006
~069108 MULL.W #1838,D2
~06910C SUB.L D2,D3
~06910E MOVE.L (A1),D4
~069110 LSR.L #2,D4
~069112 SUBQ.L #1,D4
~069114 BSR 0006911E
~069118 BSR 00068FB2
~06911C BRA 00069114
=====
λ 06911E MOVE.W #F,D2
~069122 MOVE.L #1860,D0
~069128 MOVEA.L 691F8(PC),A6
~06912C MOVE.W #2,00DFF09C
~069134 MOVE.L A6,00DFF020
~06913A MOVE.W #8210,00DFF096
~069142 MOVE.W #4489,00DFF07E
~06914A MOVE.W #7F00,00DFF09E
~069152 MOVE.W #B500,00DFF09E
~06915A MOVE.W #4000,00DFF024
~069162 MOVE.B 00BFD000,D1
~069168 MOVE.B 00BFD000,D1
~06916E BTST #4,D1
~069172 BEQ 00069168

```

Figure 3m – more loader code

```

d 69088
~069088 MOVE.B #7D,00BFD100
~069090 NOP
~069092 NOP
~069094 MOVE.B #75,00BFD100
~06909C MOVE.W #B000,D1
~0690A0 DBF D1,000690A0
~0690A4 RTS
=====
λ 0690A6 MOVE.B #79,00BFD100
~0690AE NOP
~0690B0 NOP
~0690B2 MOVE.B #71,00BFD100
~0690BA MOVE.W #B000,D1
~0690BE DBF D1,000690BE
~0690C2 RTS

```

Figure 3n - The two new routines in this loader

```

d 69006
~069006 CMP.W    691FC(PC),D0
~06900A BEQ      0006901E
~06900C BGT      00069022
~06900E MOVE.W   691FC(PC),D6
~069012 SUB.W    D0,D6
~069014 SUBQ.W   #1,D6
~069016 BSR      00068FDC
~069018 DBF      D6,00069016
~06901C RTS

=====
~06901E BSR      00068FB2
~069020 BRA      00068FDC
=====
~069022 SUB.W    691FC(PC),D0
~069026 SUBQ.W   #1,D0
~069028 BSR      00068FB2
~06902A DBF      D0,00069028
~06902E RTS

=====
~069030 MOVE.B   00BFE001,D0
~069036 BTST     #4,D0
~06903A BEQ      00069044
~06903C BSR      00068FDC
~06903E BSR      00069056

```

Figure 3o – more differences in the second loader – the track seek routine.

```

Search from: 068000 to: 069200
068130 BSR      000690C4
068140 BSR      000690C4
068180 BSR      000690C4
068280 BSR      000690C4
068394 BSR      000690C4
068470 BSR      000690C4
06848A BSR      000690C4
0684A0 BSR      000690C4
Searched up to adr: 069200
Ready.
d 68122
~068122 BSR      00068F5E
~068126 LEA      00000500,S,A0
~06812A LEA      00069200,A1
~068130 BSR      000690C4
~068134 LEA      0006C780,A0
~06813A LEA      00069200,A1
~068140 BSR      000690C4
d 68270
~068270 DBF      D0,000681F4
~068274 LEA      0006C780,A0
~06827A LEA      00069210,A1
~068280 BSR      000690C4

```

Figure 4a - Calls to load routine (1)

```

d 68384
~068384 DBF      D0,00068364
~068388 LEA      00032000,A0
~06838E LEA      00069220,A1
~068394 BSR      000690C4
~068398 BSR      00069064
~06839C TST.W    00067FFE
d 68464
~068464 LEA      00027620,A0
~06846A LEA      00069220,A1
~068470 BSR      000690C4
~068474 LEA      0001E986,A1
~06847A BSR      00068EA8
~06847E LEA      00027620,A0
~068484 LEA      00069230,A1
~06848A BSR      000690C4
~06848E MOVEA.L  A0,A1
d 68494
~068494 LEA      00058F00,A0
~06849A LEA      00069230,A1
~0684A0 BSR      000690C4
~0684A4 MOVEA.L  A0,A1
~0684A6 BSR      00068EA8

```

Figure 4b – calls to load routine (2)

```

n 69200
:069200 00 00 24 4C 00 03 13 D0 00 03 38 1C 00 00 2E CC ..$L.....8.....
:069210 00 03 66 E8 00 00 48 EC 00 03 AF D4 00 00 6C E0 ..f...H.....l.
:069220 00 04 1C B4 00 01 F4 00 00 06 10 B4 00 00 0F 64 .....d
:069230 00 06 20 18 00 00 9E 84 00 06 BE 9C 00 00 10 A4 ..

```

Figure 4c - File table area

```

a 240000
^240000 move.l (a1),d2
^240002 sub.l #1838,d2
^240008 add.l #280000,d2
^24000E move.l 4(a1),d3
^240012 move.l d2,a2
^240014 move.b (a2)+,(a0)+
^240016 sub.l #1,d3
^24001C bne 240014
^24001E rts
^240020
d 690c4
~0690C4 MOVEA.L  A0,-(A7)
~0690C6 BSR      000690CE
~0690CA MOVEA.L  (A7)+,A0
~0690CC RTS
a 690ce
^0690CE jmp 240000
^0690D4
ln beast1_0.raw,280000
Loading from 280000 to 2F6110
Disk ok
-

```

Figure 4d - Test raw data using a memory loader

```

No known virus in memory!
Ready.
faq dff020 0 80000
Search from: 000000 to: 000000
00137C LEA      00DFF020,A0
028852 MOVE.L   00038706,00DFF020
03450E MOVE.L   00038706,00DFF020
0384E2 MOVE.L   A6,00DFF020
069134 MOVE.L   A6,00DFF020
Searched up to adr: 07FFFE
Ready.
-

```

Figure 4e – search for accesses to \$dff020

```

=====
^038472 MOVE.L   A0,-(A7)
~038474 BSR      0003847C
~038478 MOVEA.L  (A7)+,A0
~03847A RTS
=====
^03847C LEA      38308(PC),A3
~038480 TST.W    (A3)
~038482 BNE      00038488
~038484 BSR      0003830C
~038488 MOVE.L   (A1)+,D0
~03848A CMP.L    #822D0,D0
~038490 BGE      00038496
~038492 BSR      00038436
~038494 BRA      00038498
=====
^038496 BSR      00038454
~038498 MOVE.L   D0,D3
~03849A DIVU.W   #1838,D0
~03849E CMP.W    #56,D0
~0384A2 BLT      000384AE
~0384A4 SUBI.L   #822D0,D3
~0384AA SUBI.W   #56,D0
~0384AE MOVEQ    #0,D2
~0384B0 MOVE.W   D0,D2
=====

```

Figure 4f – third load routine

```

02E9BC JSR      00038472
02E9D6 JSR      00038472
02EAAA JSR      00038472
03032E JSR      00038472
03034C JSR      00038472
03051C JSR      00038472
030536 JSR      00038472
030566 JSR      00038472
030F56 BSR      00038472
030F6E BSR      00038472
030FF6 BSR      00038472
03101E BSR      00038472
031040 BSR      00038472
03109A BSR      00038472
0310B2 BSR      00038472
03119C BSR      00038472
0311D6 BSR      00038472
0312E0 JSR      00038472
0314D8 JSR      00038472
031834 BSR      00038472
036366 BSR      00038472
03781C JSR      00038472
Searched up to adr: 07FFFE
Ready.

```

Figure 4g – calls to third loader

```

d 2e9cc
~02E9CC LEA    00000500,S,A0
~02E9D0 LEA    0003868E,A1
~02E9D6 JSR    00038472
d 2ea9e
~02EA9E LEA    0005DA00,A0
~02EAA4 LEA    000386FE,A1
~02EAAA JSR    00038472
d 30323
~030322 LEA    0004A950,A0
~030328 LEA    000385FE,A1
~03032E JSR    00038472
d 30340
~030340 LEA    00050000,A0
~030346 LEA    000385CE,A1
~03034C JSR    00038472
d 30510
~030510 LEA    00044E00,A0
~030516 LEA    0003860E,A1
~03051C JSR    00038472
d 3052a
~03052A LEA    00046300,A0
~030530 LEA    00038616,A1
~030536 JSR    00038472
-

```

Figure 4h – some example calls to loader

```

:03859E 20 C0 51 CA FF E8 4E 75 00 00 00 00 00 00 2F A0 .Q...Nu...../.
:0385AE 00 00 2F A0 00 00 04 BC 00 00 34 5C 00 00 36 D0 ../.....4\..6.
:0385BE 00 00 6B 2C 00 00 08 14 00 00 73 40 00 01 E4 88 ..k.....s0..ä.
:0385CE 00 02 57 C8 00 00 03 DC 00 02 5B A4 00 00 2E 9C ..W...ü..[.....
:0385DE 00 02 8A 40 00 00 64 D0 00 02 EF 10 00 00 35 B8 ..0..d.....5.
:0385EE 00 03 24 C8 00 00 02 94 00 03 27 5C 00 00 B4 BC ..$......\....
:0385FE 00 03 DC 18 00 00 26 C0 00 04 02 D8 00 00 4F 3C ..ü...&.....0<
:03860E 00 04 52 14 00 00 08 6C 00 04 5A 80 00 00 C7 40 ..R...l..Z....@
:03861E 00 05 21 C0 00 00 56 44 00 05 78 04 00 01 99 40 ..!...vD..x....@
:03862E 00 07 11 44 00 00 73 10 00 07 84 54 00 00 07 14 ...D..s...T....
:03863E 00 08 3B DC 00 00 9E D4 00 08 DA B0 00 00 25 90 ;;ü.....%.
:03864E 00 09 00 40 00 00 70 5C 00 09 70 9C 00 00 5C C8 ...@..p\..p...\
:03865E 00 09 CD 64 00 00 69 B8 00 0A 37 1C 00 00 2A 24 ...d..i...7...*$
:03866E 00 0A 61 40 00 00 3D 5C 00 0A 9E 9C 00 00 05 D0 ...a@..=\.....
:03867E 00 0A A4 6C 00 00 BE 74 00 0B 62 E0 00 00 50 A4 ...l...t..b...P.
:03868E 00 0B B3 84 00 01 E2 C8 00 0D 96 4C 00 00 5D AC .....L...l.
:03869E 00 0D F3 F8 00 00 31 F8 00 0E 25 F0 00 00 3E 00 .....1...%.>.
:0386AE 00 0E 63 F0 00 01 31 98 00 08 3B DC 00 00 50 A4 ..c...l...;ü..P.
:0386BE 00 08 8C 80 00 00 69 58 00 08 F6 00 00 00 2D D4 .....iX...ö...-.
:0386CE 00 09 23 D4 00 00 1F A8 00 09 43 7C 00 00 63 80 ..#.....C|..c.
:0386DE 00 09 A6 FC 00 00 03 00 00 09 A9 FC 00 00 3D 5C ...ü.....ü..=\
:0386EE 00 09 E7 58 00 01 D2 B4 00 0B BA 0C 00 00 13 28 ...X.....<
:0386FE 00 00 19 0C 00 00 04 00 00 00 00 00 00 00 00 ..
:03870E 00 00 00 00 6D D7 EF 08 00 40 08 2F 37 35 4A 40 ...n....@./75J@
:03871E 48 20 3F BE AF FB FF FE 5B 5F 1B FF F0 E0 BF FE H ?.....L_.....

```

Figure 4i – the whole file table (with some spurious data before and after).

Now we can search for additional load routines by searching for accesses to DFF020. The results of this can be seen in figure 4e. There are several accesses but the one at \$69134 was the one we saw previously and the one at \$384e2 looks very similar. Scrolling back through the disassembly at this address we see another (third) loader that looks the same as the previous one. (See figure 4f)

Using FAQ to find calls to the third loader we see many calls (some shown in figure 4g). If we disassemble some of these calls we can find that the load table used here (See figure 4h). If we look at the memory around these addresses we see what looks like the whole file table (see figure 4i). It seems to stretch between \$385a6 and \$38706.

The contents of this seem to be:

Address	Disk start offset	Length
385a6	0	2fa0
385ae	2fa0	4bc
385b6	345c	36d0
385be	6b2c	814
385c6	7340	1e488
385ce	257c8	3dc
385d6	25ba4	2e9c
385de	28a40	64d0
385e6	2ef10	35b8
385ee	324c8	294
385f6	3275c	b4bc
385fe	3dc18	26c0
38606	402d8	4f3c
3860e	45214	86c
38616	45a80	c740
3861e	521c0	5644
38626	57804	19940
3862e	71144	7310
38636	78454	714
3863e	83bdc	9ed4
38646	8dab0	2590
3864e	90040	705c
38656	9709c	5cc8
3865e	9cd64	69b8
38666	a371c	2a24
3866e	a6140	3d5c
38676	a9e9c	5d0
3867e	aa46c	be74
38686	b62e0	50a4
3868e	bb384	1e2c8
38696	d964c	5dac
3869e	df3f8	31f8
386a6	e25f0	3e00
386ae	e63f0	13198
386b6	83bdc	50a4
386be	88c80	6958
386c6	8f600	2dd4

386ce	923d4	1fa8
386d6	9437c	6380
386de	9a6fc	300
386e6	9a9fc	3d5c
386ee	9e758	1d2b4
386f6	bba0c	1328
386fe	190c	4

If we look at this table we can see 3 apparent parts. One part starts at \$385ae and ending at \$386ae. The disk start values in this section range from \$2fa0 and \$e63f0. This pretty much covers both sides of the disk if we consider the start of side 1 starts at \$822d0. These are likely the file locations for disk 2. The second part of the table runs between \$386b6 and \$386f6. The disk start values in this section are between and \$83bdc and \$bba0c. These are all on side 1 and are probably the remaining file locations for disk 1 (as we saw previously that most of side 0 was used in the previous loader routine). Finally there is one last entry that seems to sit alone (at \$386fe).

Looking at the file tables and calculating the amount of file data on each disk (using our guesses as to which parts of the file table are for which disk). If we total up the space used on disk 1 and disk 2 we end up with values approximately 655k used on disk 1 and 950k of data on disk 2. Clearly disk 2 is going to be a major stumbling block for us. Seems like we have two possible options

- Move some of the data from disk 2 to disk 1 (the total amount of data should still fit on two AmigaDOS 880k disks). This will probably require additional disk swapping and also for us to write some code which will advise the user when to swap disks.
- Attempt to compress the data and see if it can be stored as it was on the original, this would be the ideal solution if possible.

Either way we need to extract all of the files individually so that we can progress.

Extracting the individual files

We have the file tables and the extracted MFM data. We can use these to get the individual files we need. We can either do this manually using AR and the LM and SM commands for each file after calculating the addresses in memory based on the file table. The other option would be to write some code to do this automatically.

This code is something that as a cracker 'back in the day' would be most useful. This could be a small piece of code that incbins the MFM image data and the file table and loops through the file table saving out the files using the dos.library functions. Ideally you would have an Amiga with a hard disk so that the files can be easily stored. This kind of program can quickly be altered to handle whatever format the file table might be in and can be reused to help work on many different cracks. In addition where there are many files to deal with its more efficient than the manual method in AR.

In our case we can just do it manually loading up the image data and saving out the files to a blank disk. We will need 3 disks to store them. (see Figure 5a – 5f). Don't forget the image data doesn't all start at offset 0 (only disk 2 side 0 does – the others start at track 1). Also notice how we calculate the memory load address to make it easier to pretty much copy out the values from the file table directly.

Now that we have a copy of all of the files we need to repack them to see if we can now fit them onto the 2 disks they originally came from. We've already seen the unpack routine based on Bytekiller during our investigation. We can probably assume that the same packer will be used throughout but it's a good idea to check to see that is the case. Let's boot the original disk 1 and stop it when the Psygnosis screen first appears. Put a breakpoint at \$690cc (the end of the load routine) and let it continue until the breakpoint is hit. You should end up with something like figure 5h. The exact addresses may be different if you aren't quick enough when the title page loads. You should use the ST command to step after the RTS of the load routine and see what the next few lines of code are. You will probably see something along the lines of what is shown in figure 5h. It takes the load address that was in A0 and copies it to A1. You may also see a different value being placed in A1 if you are viewing a different address later on in the load process (some of the files are unpacked over themselves others are unpacked to a different address). The BSR \$68ea8 is the call to the unpack routine. If you take a look at this routine you will see it is the same as the one we saw previously (see figure 5i).

If you look at each of the files we have extracted in a hex editor at the end of the file you will see a particular string that reads DATA COMPACTION ©1989 A.R.BOND this is the signature of the packer (see figure 5j). We know this is a derivative of Bytekiller and we need a quick way of unpacking these files. We could extract the unpack routine and write something to run through all the files and checking for the signature and unpacking them as needed. However luckily for us in this case there is a program called XFDMaster that can recognise and unpack many different file formats. This would not have been available to the original crackers of this game and they would most likely have had to unpack the files manually or at least write something to unpack them. In the interest of speed and to save my fingers I think it makes sense to use this program since one of the formats it recognises is one called BOND packer which is the one we are interested in.

I won't go into the intricate details of how to run XFDMaster. If you are able to handle the rest of this tutorial then you should be able to figure out how to setup and use XFDMaster. It's probably easiest to use if you have an Amiga with a HD. Once you have used the command line tool XFDDecrunch (part of the XFDMaster tool) to unpack the files you will see that not all of them successfully unpacked. This suggests that some of them are not packed or that they are packed differently somehow that XFDMaster is not able to unpack.

Here is a list of the files that don't unpack using XFDMaster:

Disk1:

Beast_00244c

Beast_041cb4

Beast_0610b4

Beast_062018

Beast_088c80

Beast_09e758

Disk2:

Beast_007340

Beast_057804

Beast_0a6140

Beast_0bb384

Beast_0d964c

Beast_0e63f0

Looking at the files in a hex editor (or loading them into AR and viewing the memory) it seems clear that some of the files are not packed (they have many areas of blank space – e.g. 0's).

Disk1:

Beast_00244c (contains obviously uncompressed data)

Beast_041cb4 (some uncompressed data at the end)

Beast_0610b4 (looks like compressed data – signature at the end is incomplete)

Beast_062018 (looks like compressed data – signature is followed by some additional bytes)

Beast_088c80 (looks like compressed data –signature is followed by some 0 bytes)

Beast_09e758 (contains obviously uncompressed data)

Disk2:

Beast_007340 (contains obviously uncompressed data)

Beast_057804 (contains obviously uncompressed data)

Beast_0a6140 (looks like compressed data but no signature at all)

Beast_0bb384 (contains obviously uncompressed data)

Beast_0d964c (looks like compressed data – signature looks corrupted)

Beast_0e63f0 (contains obviously uncompressed data)

So we have 5 files in total that look to be compressed but don't contain the correct signature.

Disk1:

Beast_0610b4

Beast_062018

Beast_088c80

Disk2:

Beast_0a6140

Beast_0d964c

If we look at the unpack routine we see the first thing it does is read the first long word from the packed data and add it onto the start of A0 which contains the packed data address. So this longword is an offset to the end of the packed data and then it reads a longword 4 bytes backwards from that address and puts it into A2 and adds A1. This will be the end of the unpacked data (so the value read is the length of the unpacked data). So we can look at the files we aren't sure about and see if the first longword looks like an offset to the end of the packed data and then the longword at the 4 bytes prior to that offset looks like an unpacked file size. If these files match that pattern we have a good bet they are packed.

What we need to do then is to unpack these files manually using the unpack routine. We just need to load these files to a fixed address using LM and then write a small piece of code to put the relevant addresses into A0 and A1 and call the unpack routine. (see figure 5k and 5l). Repeat this process for each of the files and we now have a complete set of unpacked files. Now gather up all of the now unpacked files and the files that weren't packed in the first place and you should end up with something along the lines of what's shown in figure 5m and 5n).

We now need to repack these files – I intend to use RNC Propack as it offers good efficiency and an acceptable unpack speed on the A500. See figure 5o and 5p for the results of the packing. This looks extremely promising. The disk 1 files compress to around 540k and the disk 2 files compress to 780k. Based on these sizes it will be no problem to get the files onto the disks as they were originally organised.

Based on these figures it seems entirely possible that we can create a standard AmigaDOS disk and save the files onto them and use a file loader rather than a track loader. This approach will mean that we don't have to mess around writing the raw sectors back to disk and mess around with the file table to adjust according to the new file sizes. We can simply

use the disk start offset to figure out which file we need to load and get the size from the file and ignore the size in the file table.

We already have the files in a suitably named format so we can simply copy them onto 2 standard AmigaDOS disks and then we need to figure out how to patch in a new load routine and a new unpack routine.

```
In beast1_0.raw, 301838
Loading from 301838 to 377948
Disk ok
sm beast_00244c, 30244c 33381c
Disk ok
sm beast_03381c, 33381c 3366e8
Disk ok
sm beast_0366e8, 3366e8 33afd4
Disk ok
sm beast_03afd4, 33afd4 341cb4
Disk ok
sm beast_041cb4, 341cb4 3610b4
Disk ok
sm beast_0610b4, 3610b4 362018
Disk ok
sm beast_062018, 362018 36be9c
Disk ok
sm beast_06be9c, 36be9c 36cf40
Disk ok
_
```

Figure 5a – save file data for disk 1 side 0

```
?022d0+1838
%00000000000010000011101100001000 = $00083B08 = !0000539400

In beast1_1.raw, 283b08
Loading from 283B08 to 2F9C18
Disk ok
sm beast_083bdc, 283bdc 288c80
Disk ok
sm beast_088c80, 288c80 28f600
Disk ok
sm beast_08f600, 28f600 2923d4
Disk ok
sm beast_0923d4, 2923d4 29437c
Disk ok
sm beast_09437c, 29437c 29a6fc
Disk ok
sm beast_09a6fc, 29a6fc 29a9fc
Disk ok
sm beast_09a9fc, 29a9fc 29e758
Disk ok
sm beast_09e758, 29e758 2bba0c
Disk ok
sm beast_0bba0c, 2bba0c 2bcd34
Disk ok
```

Figure 5b – save file data for disk 1 side 1

```

In beast2_0.raw, 300000
Loading from 300000 to 379180
Disk ok
sn beast_000000, 300000 302fa0
Disk ok
sn beast_002fa0, 302fa0 30345c
Disk ok
sn beast_00345c, 30345c 306b2c
Disk ok
sn beast_006b2c, 306b2c 307340
Disk ok
sn beast_007340, 307340 3257c8
Disk ok
sn beast_0257c8, 3257c8 325ba4
Disk ok
sn beast_025ba4, 325ba4 328a40
Disk ok
sn beast_028a40, 328a40 32ef10
Disk ok
sn beast_02ef10, 32ef10 3324c8
Disk ok
sn beast_0324c8, 3324c8 33275c
Disk ok

```

Figure 5c - save file data for disk 2 side 0 (1)

```

sn beast_03275c, 33275c 33dc18
Disk ok
sn beast_03dc18, 33dc18 3402d8
Disk ok
sn beast_0402d8, 3402d8 345214
Disk ok
sn beast_045214, 345214 345a80
Disk ok
sn beast_045a80, 345a80 3521c0
Disk ok
sn beast_0521c0, 3521c0 357804
Disk ok
sn beast_057804, 357804 371144
Disk ok
sn beast_071144, 371144 378454
Disk ok
sn beast_078454, 378454 378b68
Disk ok

```

Figure 5d – save file data for disk 2 side 0 (2)

```

In beast_2_1.raw, 283b08
Loading from 283B08 to 2F9C18
Disk ok

sn beast_083bdc, 283bdc 28dab0
Disk ok
sn beast_08dab0, 28dab0 290040
Disk ok
sn beast_090040, 290040 29709c
Disk ok
sn beast_09709c, 29709c 29cd64
Disk ok
sn beast_09cd64, 29cd64 2a371c
Disk ok
sn beast_0a371c, 2a371c 2a6140
Disk ok
sn beast_0a6140, 2a6140 2a9e9c
Disk ok
sn beast_0a9e9c, 2a9e9c 2aa46c
Disk ok

```

Figure 5e – save file data for disk 2 – side 1 (1)

```
sn beast_0aa46c, 2aa46c 2b62e0  
Disk ok  
sn beast_0b62e0, 2b62e0 2bb384  
Disk ok  
sn beast_0bb384, 2bb384 2d964c  
Disk ok  
sn beast_0d964c, 2d964c 2df3f8  
Disk ok  
sn beast_0df3f8, 2df3f8 2e25f0  
Disk ok  
sn beast_0e25f0, 2e25f0 2e63f0  
Disk ok  
sn beast_0e63f0, 2e63f0 2f9588  
Disk ok
```

Figure 5f – save file data for disk 2 – side 1 (2)



Figure 5g – Psygnosis loading screen

```

No known virus in memory!
Ready.
bs 690cc
Breakpoint inserted
Ready.
X
No known virus in memory!
Ready.
Breakpoint raised at address: 000690CC
r
D0=4F4E443C 45444414 0000FFFF 00000000 000005E0 00000000 FFFFFFFF 0000FFFF
A0=000763C0 00069214 0006C780 000691FE 00002E20 000018B6 00054310 000003FC
PC = 000690CC USP = 000018B2 SR = 2010 T=0 S=1 I=000 X=1 N=0 Z=0 V=0 C=0
d 690cc
~0690CC RTS
=====
^0690CE LEA 691FE(PC),A3
st
D0=4F4E443C 45444414 0000FFFF 00000000 000005E0 00000000 FFFFFFFF 0000FFFF
A0=000763C0 00069214 0006C780 000691FE 00002E20 000018B6 00054310 00000400
PC = 00068184 USP = 000018B2 SR = 2010 T=0 S=1 I=000 X=1 N=0 Z=0 V=0 C=0
~068184 MOVEA.L A0,A1
~068186 BSR 00068EA8
~06818A TST.W 00067FFE

```

Figure 5h – breakpoint end of loader and see following code

```

d 68ea8
~068EA8 ADDA.L (A0),A0
~068EAA MOVEA.L -(A0),A2
~068EAC ADDA.L A1,A2
~068EAE MOVE.L -(A0),D5
~068EB0 MOVE.L -(A0),D0
~068EB2 EOR.L D0,D5
~068EB4 LSR.L #1,D0
~068EB6 BNE 00068EBA
~068EB8 BSR 00068F2C
~068EBA BCS 00068EEE
~068EBC MOVEQ #8,D1
~068EBE MOVEQ #1,D3
~068EC0 LSR.L #1,D0
~068EC2 BNE 00068EC6
~068EC4 BSR 00068F2C
~068EC6 BCS 00068F14
~068EC8 MOVEQ #3,D1
~068ECA CLR.W D4
~068ECC BSR 00068F38
~068ECE MOVE.W D2,D3
~068ED0 ADD.W D4,D3
~068ED2 MOVEQ #7,D1
~068ED4 LSR.L #1,D0

```

Figure 5i – unpack routine

```

In beast 08f600,40000
Loading From 040000 to 042DD4
Disk ok
n 42d00
:042D00 53 28 68 05 03 00 03 01 1B 01 00 C1 83 40 5F C0 S(h.....@_
:042D10 08 20 E0 0A 05 06 2C 02 01 3A 01 00 BB 80 18 18 . ....P`
:042D20 18 18 05 04 38 04 02 10 02 80 50 60 85 05 00 02 ...8....P`
:042D30 40 20 10 37 20 E0 6D 80 10 18 00 70 0F F0 07 04 @ ,7 .n...p...
:042D40 80 A0 60 20 10 40 10 08 30 08 04 28 04 02 4A 02 .'.@..0..(.J.
:042D50 01 10 04 20 30 32 10 10 0E F0 08 06 C0 04 03 86 ... 02.....
:042D60 02 01 FE 05 00 A7 80 80 A0 20 17 2F F6 2F 9F ED ....s.../o/..
:042D70 C8 10 98 1C 70 30 B9 2C BD 61 46 C0 E3 41 D0 00 ....p0...aF..A..
:042D80 0E 1E DA 04 97 20 49 72 04 96 5E B0 A3 78 00 20 .... Ir..^..x.
:042D90 1E 00 10 0B 00 08 04 80 04 02 80 08 00 FF 70 01 .....P.
:042DA0 E7 60 00 10 37 5D E5 59 00 00 76 20 00 01 03 F5 ...7l.V..v....
:042DB0 4D 45 3E 44 41 54 41 20 43 4F 4D 50 41 43 54 49 ME>DATA COMPACTI
:042DC0 4F 4E 20 28 43 29 31 39 38 39 20 41 2E 52 2E 42 ON (C)1989 A.R.B
:042DD0 4F 4E 44 3C 02 7F EF D3 5F DB FF DF FF FD FF FF OND<...B...

```

Figure 5j – unpack signature

```

rt 0 1 40000
Disk ok
d 40070
~040070 ADDA.L (A0),A0
~040072 MOVEA.L -(A0),A2
~040074 ADDA.L A1,A2
~040076 MOVE.L -(A0),D5

In beast 0610b4, 50000
Loading from 050000 to 050F64
Disk ok
a 30000
^030000 lea 50000,a0
^030006 lea 51000,a1
^03000C jsr 40070
^030012 bra 30012
^030014
bs 30012
Breakpoint inserted
Ready.
g 30000
No known virus in memory!
Ready.
Breakpoint raised at address: 00030012

```

Figure 5k – manually unpack the remaining files

```

n 50000
:050000 00 00 0F 40 62 3F E4 45 74 7F 3D EA 80 80 A0 7F ...0b?3Et*=....
n 50f3c
:050F3C 00 00 15 E0 01 01 03 F5 4D 45 3E 44 41 54 41 20 .....ME>DATA
sm beast_0610b4.unpack, 51000 525e0
Disk ok

```

Figure 5l – save unpacked data


```

Packing Data file(s)
beast_0e63f0 [ 78232] [ 60529] [22.62%]
beast_0e25f0 [ 40000] [ 13629] [63.92%]
beast_0df3f8 [ 40000] [ 10844] [72.89%]
beast_0d964c [ 48876] [ 20635] [57.78%]
beast_0bb384 [ 123592] [ 94918] [52.32%]
beast_0b62e0 [ 43144] [ 17823] [58.68%]
beast_0aa46c [ 76800] [ 41061] [46.53%]
beast_0a9e9c [ 5376] [ 1090] [79.72%]
beast_0a6140 [ 22080] [ 10490] [52.49%]
beast_0a371c [ 16870] [ 9365] [44.48%]
beast_09cd64 [ 50616] [ 23233] [54.09%]
beast_09709c [ 30000] [ 21381] [28.73%]
beast_090040 [ 50554] [ 21056] [58.34%]
beast_08dab0 [ 16790] [ 8784] [47.68%]
beast_083bdc [ 76800] [ 32854] [57.22%]
beast_078454 [ 5376] [ 1307] [25.68%]
beast_071144 [ 47914] [ 25645] [46.47%]
beast_057804 [ 104768] [ 84591] [19.25%]
beast_0521c0 [ 39086] [ 18662] [52.25%]
beast_045a80 [ 76800] [ 42062] [45.23%]
beast_045214 [ 5376] [ 1702] [68.34%]
beast_0402d8 [ 22080] [ 18898] [14.41%]
beast_03dc18 [ 16683] [ 9167] [45.05%]
beast_03275c [ 67500] [ 42190] [37.49%]
beast_0324c8 [ 5376] [ 529] [90.15%]
beast_02ef10 [ 30240] [ 10894] [63.97%]
beast_028a40 [ 50300] [ 22211] [55.84%]
beast_025ba4 [ 16692] [ 9293] [44.32%]
beast_0257c8 [ 1472] [ 871] [40.82%]
beast_007340 [ 124040] [ 98973] [20.20%]
beast_006b2c [ 3456] [ 1856] [46.29%]
beast_00345c [ 24000] [ 12126] [49.47%]
beast_002fa0 [ 2520] [ 970] [61.50%]
beast_000000 [ 45108] [ 9064] [79.90%]

34 file(s) [1408517] [ 798703] [12.80%] [0h 0m 14s]

```

Figure 5p – disk 2 – repacked RNC propack

Patching the load and unpack routines.

Now that we have all the files extracted and repacked we need to hook in a new load and depack routine. First of all we need a file based loader routine. Mine is an extension of my track based load routine – however it requires a 512 byte work area in addition to the MFM buffer. It could be improved to remove this requirement but in this case the MFM buffer used by the game is large enough that we can use the MFM buffer for both buffers required when using my loader. My routine expects the following:

- A0 – load address
- A1 – filename
- A2 – 512 byte work area
- A3 – MFM buffer address
- D0 – drive number

The routine is larger than my track load routine at 1176 bytes so we will need to find spare memory for that routine and in addition we will need to find room for the RNC unpack code which is another 560 bytes.

In addition we also need to consider that not all of the original files were packed and we have now packed these so we will need to allow for this when we hook up our depack routine. We need to patch in our RNC unpack code into the existing unpack code but we also need to force any files to unpack that were not originally packed (where there won't be any calls to the original unpack code). I suggest that any files that were not originally packed can be unpacked immediately after the load routine completes and can be unpacked in

place in memory (e.g. at the same address they were loaded to). We need to leave the unpacking of any files that were originally packed to the patched unpack routine however as they will sometimes be unpacked to a different address. To achieve this I will set a flag to indicate which files need to be unpacked by the load routine. The solution I came up with to store this flag is to use the highest bit in the disk start offset in the file table. When this bit is set the loader will automatically unpack the file.

So we know that the two parameters passed into the original load routine are the file table entry in A0 and the load address in A1. We need to find out where the MFM buffer is for this routine and we can use the same address for our MFM buffer. Once we have that information we can put together a small piece of interface code that sets up the registers as required by our load routine and loads the file (and unpacks it if necessary).

If we load up the previously saved loader to \$68000 (see figure 6a) and disassemble the load routine at \$690c4 we can see where it sets the \$dff020 register which is the DSKPTH register (see figure 6b) we saw this piece of code previously when searching for the load routine. So now we can see the address of the MFM buffer is held in \$691f8. We can now build our interface code.

If you look at figure 6c and 6d you will see that it stores a0/a1 to the stack for later use then calls the main start of the loader interface code. When that is done it pulls back the old a0/a1 and tests bit 7 of the first byte at (a1) this is the flag that we use to signal to the loader to do an unpack. If this bit is not set it simply returns, otherwise it puts the load address into A1 and calls what will be the unpack routine. When the unpack routine returns this will be the end of the load.

The main bulk of the interface code pulls out the disk start offset at (a1) and the MFM buffer address we previously discovered into A3. It also sets up A2 which is the 512 byte work area and points it to the end of the MFM buffer area. Then it calculates the filename based on the disk start address (by using a lookup table for the hex codes 0-f and plugging them into a filename string one hex digit at a time) and loads the filename pointer into A1. Then it calls \$4006c which will be our main loader code.

Eventually we will need to find room to put these routines in memory but for now we can put them somewhere in fast memory where they will not interfere with the game.

Firstly we need to update the file table in the load routine we have saved to set the high bits for the files that need to be unpacked by the loader (see figure 6e – the two files on disk 1 side 0 have been flagged by setting high bit at \$69200 and \$69220).

Load up the file loader and unpack routines to \$280000 and \$290000 (this is fast memory in my setup) and make sure the load routine is jumping to the unpack routine correctly. Set up the load routine at \$68000 and the code to go to supervisor mode at \$50000 (see figure 6f). Hook in the load routine and unpack routines we saw earlier so that they call the new code in fast memory (see figure 6g). Put the disk 1 on containing all the RNC packed files and run the code using G 50000

If all has gone well you will see the title page start to load and then it will start to load and then at that point it will crash horribly. It took me a while to figure out what the problem was here and some amounts of stepping through and debugging – of which I won't go into too much details regarding. Suffice to say that the code at \$68174 (see figure 6h) loads a file (beast_00244c) into memory starting at address \$763c0 and then unpacks it over itself. When the file unpacks (using the original Bytekiller unpack) it runs right up to \$7ffff. When RNC attempts to unpack it overwrites a few additional bytes at \$80000-\$80002. This causes the machine to crash (I guess if you have 1mb chip then its fine). The very simple solution is to unpack this file. There is no code changes required to handle this as the RNC unpack code simply skips if it doesn't find the unpack header.

So after you unpack beast_00244c and then repeat the steps in figure 6g again and restart using G 50000 it will this time load right through up to the insert disk 2 screen. We know from before that at this point it has loaded another load routine into memory so we need to hook our load and depack routines into there too. We saw the load routine at \$38472 and if we look through some of the calls to that routine that we saw back in figure 4g we see many of those loader calls are followed by a call to \$28960 (see figure 6i) and if we look at that we see the usual unpack code. If we hook into both of these two routines (see figure 6j) and exit out of AR we can insert disk 2 and see how we get on. Press fire and you will find that disk 2 is not recognised.

The game may now have crashed or you are still stick at the insert disk 2 screen (depending on the robustness of the error checking in your loader). If you have crashed out then reboot and set up everything so that you are back at the insert disk 2 screen and jump into AR. Put a breakpoint on the load routine we just hooked and then drop back to the game and press fire. We can then see the breakpoint is hit and we see from looking in the registers and at the address pointed to by A1 (see figure 6k) that it is attempting to load the 4 bytes from \$190c. This was the 1 file that we didn't know what it did when we extracted all of the files and repacked them. If we then look at the code where this load is called from we can see that it calls the loader with a load address of \$5da00 and then does a long word compare against \$e406039f (see figure 6l) - this looks like it could well be its way of determining which disk is present. So we need to extract that data from both disks and save it out to a file.

Insert the original game and load up to the point where the insert disk 2 screen appear. Set a breakpoint at \$37828 so we can see the results of the load. Return back to the game and leave disk 1 in the drive and press fire. When the breakpoint triggers we can save out the 4 bytes for disk 1 and then drop out of AR back to the game, insert disk 2 and press fire again and repeat the save for the file on our disk 2 (see figure 6j and 6k).

If we now reboot and repeat the load process all over again as shown in figure 6f and then figure 6g and figure 6j. Once you have done all this we then need to consider the remaining files that were not originally packed and update the second file table accordingly (1 file from disk 1 and 4 files from disk 2):

Disk1:

Beast_09e758

Disk2:

Beast_007340

Beast_057804

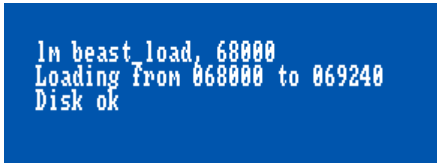
Beast_0bb384

Beast_0e63f0

After updating these files the file table should look like figure 6l.

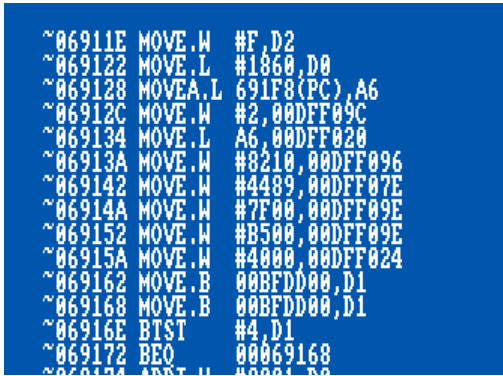
Also we need to check the MFM buffer used by the load routine at \$38472. It is not the same as the one used by the other loader. If you take a look at the routine (see figure 6m) you will see that it pulls the address of the MFM buffer from \$382fa. We need to patch our loader to take account of this now. (See figure 6n)

Once this is done we can exit AR continue loading and you should be able to see the intro screen and start loading the game at which point it gets stuck and wont progress into the game.



```
ln beast_load, 68000
Loading from 068000 to 069240
Disk ok
```

Figure 6a – load up the loader to \$68000



```
~06911E MOVE.W #F,D2
~069122 MOVE.L #1860,D0
~069128 MOVEA.L 691F8(PC),A6
~06912C MOVE.W #2,00DFF09C
~069134 MOVE.L A6,00DFF020
~06913A MOVE.W #8210,00DFF096
~069142 MOVE.W #4489,00DFF07E
~06914A MOVE.W #7F00,00DFF09E
~069152 MOVE.W #B500,00DFF09E
~06915A MOVE.W #4000,00DFF024
~069162 MOVE.B 00BFDD00,D1
~069168 MOVE.B 00BFDD00,D1
~06916E BTST #4,D1
~069172 BEQ 00069168
~069174 BTST #0,00069168
```

Figure 6b – setting up the dskpth register

```

d 40000
~040000 MOVEM.L A0-A1,-(A7)
~040004 BSR      0004001C
~040006 MOVEM.L (A7)+,A0-A1
~04000A BTST     #7,(A1)
~04000E BNE      00040014
~040010 CLR.L    D0
~040012 RTS
=====
~040014 MOVEA.L  A0,A1
~040016 JMP      00050000
=====
~04001C MOVE.L   (A1),D0
~04001E MOVEA.L  000691F8,A3
~040024 LEA      2EC0(A3),A2
~040028 LEA      40064(PC),A1
~04002C LEA      4004E(PC),A4
~040030 MOVEQ    #5,D6
~040032 MOVE.B   D0,D7
~040034 ANDI.W    #F,D7
~040038 MOVE.B   0(A4,D7.W),D7
~04003C MOVE.B   D7,0(A1,D6.W)
~040040 LSR.L     #4,D0
~040042 DBF      D6,00040032
~040046 LEA      4005E(PC),A1

```

Figure 6c – load routine interface code (1)

```

~04004A BSR      0004006C
~04004C RTS

n 4004e
:04004E 30 31 32 33 34 35 36 37 38 39 61 62 63 64 65 66 0123456789abcdef
:04005E 62 65 61 73 74 5F 30 30 30 30 30 30 00 00 00 00 beast_000000,...
-

```

Figure 6d – load routine interface code (2)

```

In beast_load, 68000
Loading from 068000 to 069240
Disk ok
n 69200
:069200 80 00 24 4C 00 03 13 D0 00 03 38 1C 00 00 2E CC ..$L.....8.....
:069210 00 03 66 E8 00 00 48 EC 00 03 AF D4 00 00 6C E0 ..f...H.....l.
:069220 80 04 1C B4 00 01 F4 00 00 06 10 B4 00 00 0F 64 .....d
:069230 00 06 20 18 00 00 9E 84 00 06 BE 9C 00 00 10 A4 .. .....

sm beast_load2, 68000 69240
Disk ok
-

```

Figure 6e – update file table for disk 1 side 0

```

lm fileload.all, 280000
Loading from 280000 to 280514
Disk ok
lm rnc_unpack.bin, 290000
Loading from 290000 to 29020C
Disk ok
d 280000
~280000 MOVEM.L A0-A1, -(A7)
~280004 BSR      0028001C
~280006 MOVEM.L (A7)+, A0-A1
~28000A BTST     #7, (A1)
~28000E BNE      00280014
~280010 CLR.L    D0
~280012 RTS
=====
^280014 MOVEA.L A0, A1
~280016 JMP      00050000
a 280016
^280016 jmp 290000
^28001C
a 50000
^050000 lea 68058, a0
^050006 move.l a0, $80
^05000C trap #0
^05000E _

```

Figure 6f – load the file load and unpack routines into memory. Update the file load routine to call the unpack routine.

```

a 690c4
^0690C4 jmp 280000
^0690CA
a 68ea8
^068EA8 jmp 290000
^068EAE
_

```

Figure 6g – hook in the load and unpack routines

```

d 68174
~068174 LEA      000763C0, A0
~06817A LEA      00069210, A1
~068180 BSR      000690C4
~068184 MOVEA.L A0, A1
~068186 BSR      00068EA8
~06818A TST.W    00067FFE

```

Figure 6h – causes a crash

```

~02EAC0 BNE      0002EA1C
d 30322
~030322 LEA      0004A950,A0
~030328 LEA      000385FE,A1
~03032E JSR      00038472
~030334 LEA      00046300,A1
~03033A JSR      00028960
~030340 LEA      00050000,A0
d 30510
~030510 LEA      00044E00,A0
~030516 LEA      0003860E,A1
~03051C JSR      00038472
~030522 MOVEA.L  A0,A1
~030524 JSR      00028960
~03052A LEA      00046300,A0
~030530 LEA      00038616,A1
d 28960
~028960 ADDA.L    (A0),A0
~028962 MOVEA.L  -(A0),A2
~028964 ADDA.L    A1,A2
~028966 MOVE.L    -(A0),D5
~028968 MOVE.L    -(A0),D0
~02896A EOR.L     D0,D5
~02896C LSR.L     #1,D0

```

Figure 6i – Looking at some of the loader calls we see calls to an unpack routine

```

a 28960
^028960 jmp 290000
^028966
a 38472
^038472 jmp 280000
^038478

```

Figure 6j – hook the loader3 and related unpack routines

```

bs 38472
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00038472
r
D0=000033CE 5514FFFF 400000F5 00000000 00002000 00000000 5555FFFF 00000000
A0=0005DA00 000386FE 00003DBA 0003830A 000050C0 00BFD500 002804EE 000004FC
PC = 00038472 USP = 00C0376A SR = 2004 T=0 S=1 I=000 X=0 N=0 Z=1 V=0 C=0
n 386fe
:0386FE 00 00 19 0C 00 00 00 04 00 00 00 00 00 00 00 00 .....

```

Figure 6k – insert disk 2 routine

```

~037810 LEA    0005DA00,A0
~037816 LEA    000386FE,A1
~03781C JSR    00038472
~037822 JSR    00038412
~037828 CMPI.L #E406039F,0005DA00
~037832 BNE    0003778E
~037836 JMP    0002A4B8
=====
^03783C LEA    0005BAC0,A0
00000000 00000000 00000000 00000000

```

Figure 6l – insert disk – compare result

```

bs 37828
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00037828
d
~037828 CMPI.L #E406039F,0005DA00
~037832 BNE    0003778E
~037836 JMP    0002A4B8
n 5da00
:05DA00 00 00 0B 0C 00 00 00 00 00 00 00 00 00 00 .....
sn beast_00190c, 5da00 5da04
Disk ok
x

```

Figure 6j – file beast_00190c for disk 1

```

Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00037828
n 5da00
:05DA00 E4 06 03 9F 00 00 00 00 00 00 00 00 00 00 ä.....
sn beast_00190c, 5da00 5da04
Disk ok
-

```

Figure 6k – file beast_00190c for disk 2

```

:0385A6 00 00 00 00 00 00 2F A0 00 00 2F A0 00 00 04 BC ...../.../.....
:0385B6 00 00 34 5C 00 00 36 D0 00 00 6B 2C 00 00 08 14 ..4\..6...k,...
:0385C6 00 00 73 40 00 01 E4 88 00 02 57 C8 00 00 03 DC ..s@..ä...W...ü
:0385D6 00 02 5B A4 00 00 2E 9C 00 02 8A 40 00 00 64 D0 ..f.....@..d.
:0385E6 00 02 EF 10 00 00 35 B8 00 03 24 C8 00 00 02 94 ...5...$.
:0385F6 00 03 27 5C 00 00 B4 BC 00 03 DC 18 00 00 26 C0 ...\.....ü...&.
:038606 00 04 02 D8 00 00 4F 3C 00 04 52 14 00 00 08 6C .....0<..R...l
:038616 00 04 5A 80 00 00 C7 40 00 05 21 C0 00 00 56 44 ..Z....@...t...VD
:038626 00 05 78 04 00 01 99 40 00 07 11 44 00 00 73 10 ...x....@...d..s.
:038636 00 07 84 54 00 00 07 14 00 08 3B DC 00 00 9E D4 ...T.....;ü...
:038646 00 08 DA B0 00 00 25 90 00 09 00 40 00 00 70 5C .....%....@..p\
:038656 00 09 70 9C 00 00 5C C8 00 09 CD 64 00 00 69 B8 ..p...\....d..i.
:038666 00 0A 37 1C 00 00 2A 24 00 0A 61 40 00 00 3D 5C ..7...*$...a@..=\
:038676 00 0A 9E 9C 00 00 05 D0 00 0A A4 6C 00 00 BE 74 .....l...t
:038686 00 0B 62 E0 00 00 50 A4 80 0B B3 84 00 01 E2 C8 ..b...P.....
:038696 00 0D 96 4C 00 00 5D AC 00 0D F3 F8 00 00 31 F8 ...L..J.....i.
:0386A6 00 0E 25 F0 00 00 3E 00 80 0E 63 F0 00 01 31 98 ...%...>...c...l.
:0386B6 00 08 3B DC 00 00 50 A4 00 08 8C 80 00 00 69 58 ...;ü..P.....iX
:0386C6 00 08 F6 00 00 00 2D D4 00 09 23 D4 00 00 1F A8 ..ö...-...#.....
:0386D6 00 09 43 7C 00 00 63 80 00 09 A6 FC 00 00 03 00 ...Cl..c....ü....
:0386E6 00 09 A9 FC 00 00 3D 5C 80 09 E7 58 00 01 D2 B4 ...ü..=\...X....
:0386F6 00 0B BA 0C 00 00 13 28 00 00 19 0C 00 00 00 04 .....(.....
sn filetable2, 385a6 38706
Disk ok

```

Figure 6l – Modified file table 2

```

~0384C2 BSR      000384CC
~0384C6 BSR      00038360
~0384CA BRA      000384C2
=====
~0384CC MOVE.W   #F,D2
~0384D0 MOVE.L   #1860,D0
~0384D6 MOVEA.L  382FA(PC),A6
~0384DA MOVE.W   #2,00DFF09C
~0384E2 MOVE.L   A6,00DFF020
~0384E8 MOVE.W   #8210,00DFF096
~0384F0 MOVE.W   #4489,00DFF07E

```

Figure 6m – mfm buffer for loader3

```

280014 MOVEA.L  A0,A1
~280016 JMP      00290000
=====
~28001C MOVE.L   (A1),D0
~28001E MOVEA.L  000382FA,A3
~280024 LEA      2EC0(A3),A2
~280028 LEA      280064(PC),A1
~28002C LEA      28004E(PC),A4

```

Figure 6n – update loader to pull MFM buffer address from same place as loader 3

Putting this all together

To save having to repeat the steps for starting the game from our crack disks every time while we figure out what's stopping it from working fully we can create some code to do all the manual work we have been doing each time. Firstly we need to find a place in the loader code we have extracted to execute the process of patching the file table 2 and load and unpack routines at \$38472 and \$28960. If we breakpoint the load routine and run through the load process up to the point where it asks for disk 2 you will find that the last call to the load routine is at \$684a0 followed by an unpack call at \$684a6 (see figure 7a). This is immediately followed by a JMP instruction. This looks like an ideal place to jump to our routine to perform the patching.

So our code to perform the full loading will be

Copy the extracted loader code to \$68000

Copy our file loader code to \$280000

Copy the unpack code to \$290000

Copy the patch routine to \$2a0000

Make the loader code call \$2a0000 instead of jumping to \$1e986. The patch routine at \$2a0000 will perform the following actions:

- Hook the load routine at \$38472
- Hook the unpack routine at \$28960
- Update the filetable2 as needed
- Update the loader code to get the MFM buffer address from \$382fa

- Jump to \$1e986.

Start the loader in supervisor mode at \$68058.

You can see how this all fits together in figures 7b – 7f (not all loader and unpack code is included). Save this whole piece of code as this will be the code that we use as the basis of our crack later. If you insert the crack disk and execute this code with G 50000 everything should execute fine.

```

~06848E MOVEA.L A0,A1
~068490 BSR 00068EA8
~068494 LEA 00058F00,A0
~06849A LEA 00069238,A1
~0684A0 BSR 000690C4
~0684A4 MOVEA.L A0,A1
~0684A6 BSR 00068EA8
~0684AA JMP 0001E986
=====
~0684B0 ADDQ.W #1,00068EA2
~0684B6 MOVE.W #1,00068EA4

```

Figure 7a – the last call to the loader before insert disk 2.

<pre> d 50000 ~050000 LEA 5005E(PC),A0 ~050004 LEA 5129E(PC),A1 ~050008 LEA 00068000,A2 ~05000E MOVE.W (A0)+,(A2)+ ~050010 CMPA.L A0,A1 ~050012 BNE 0005000E ~050014 LEA 5129E(PC),A0 ~050018 LEA 517B2(PC),A1 ~05001C LEA 00280000,A2 ~050022 MOVE.W (A0)+,(A2)+ ~050024 CMPA.L A0,A1 ~050026 BNE 00050022 ~050028 LEA 517B2(PC),A0 ~05002C LEA 519BE(PC),A1 ~050030 LEA 00290000,A2 ~050036 MOVE.W (A0)+,(A2)+ ~050038 CMPA.L A0,A1 ~05003A BNE 00050036 </pre>	<pre> Copy original loader code to \$68000 Copy our new DOS file loader to \$280000 Copy the RNC unpack code to \$290000 </pre>
---	---

Figure 7b – putting it all together (1)

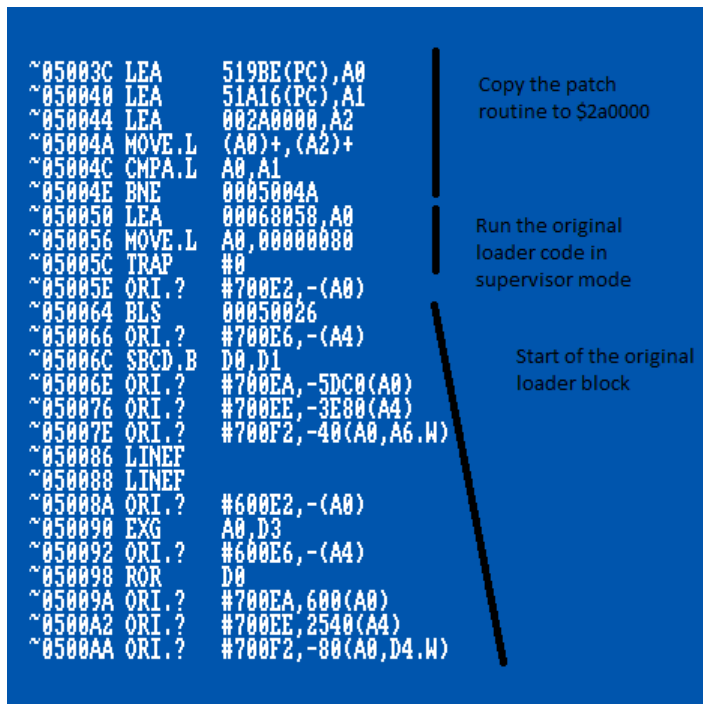


Figure 7c – putting it all together (2)

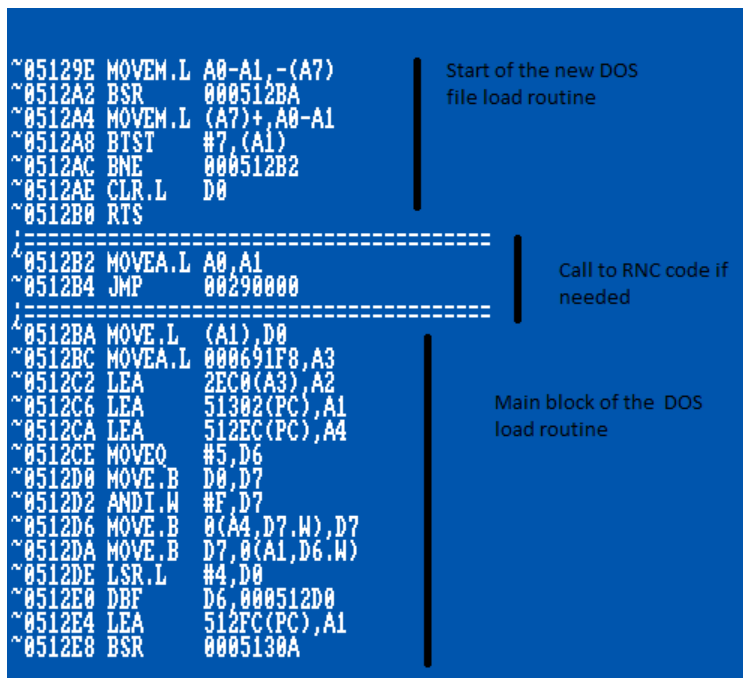


Figure 7d – putting it all together (3)

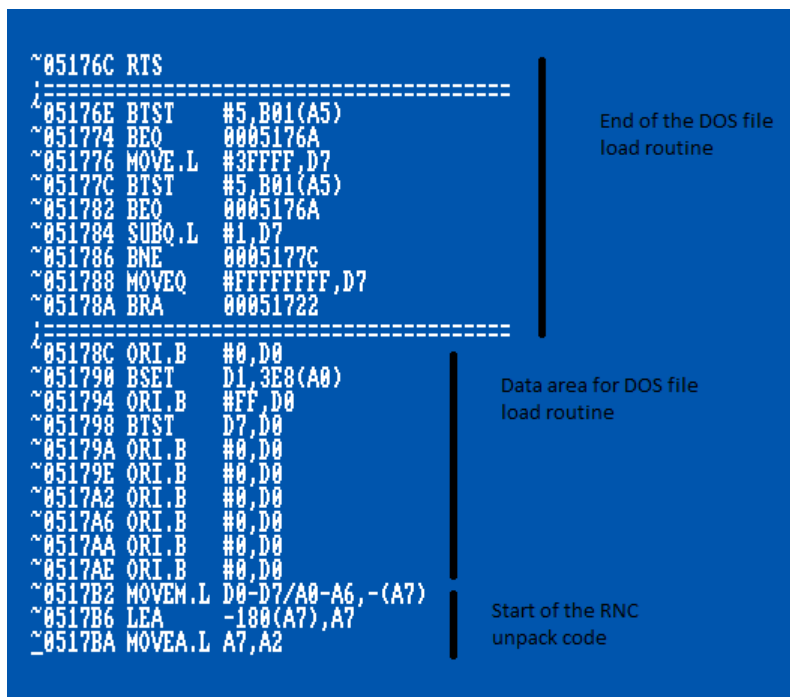


Figure 7e – putting it all together (4)

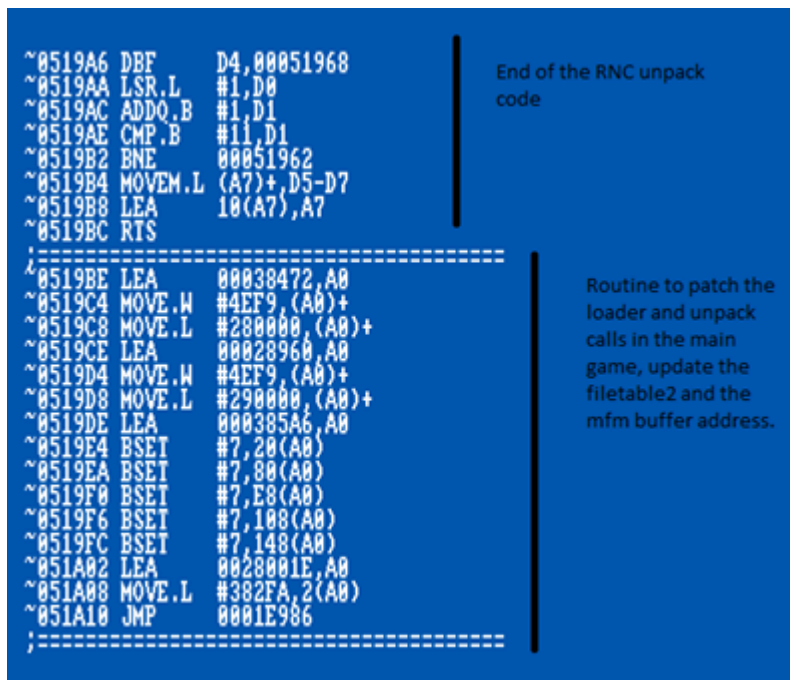


Figure 7f – putting it all together (5)

More stuff to look at

Clearly the crack is starting to look promising now. It loads past the title screen and almost into the game before freezing. Clearly there are some other things we need to address to figure out why it doesn't fully work.

If you remember way back when we were searching for code that referenced \$dff020 (back in figure 4e) there were several other routines doing this than the one we took care of. The first one was at \$137c another at \$28852 and one at \$3450e. If we go back and look at these again we will see some more stuff that needs to be taken care of. The routine at \$137c is just the original load routine from the boot block still hanging around in memory so we don't need to worry about that. The other two routines look like they are both the same.

See figure 8a to 8f for a code dump of the code starting at \$28736 which is the start of the first routine. They basically read \$3778 words of data from track 0 of the disk (more than 2 complete revolutions of the disk) and check for a value \$4454 it then finds the second instance of that value and measures how far apart they are. If they are less than \$1a2c apart then the routine returns 1 otherwise it returns 0. This is a check to ensure that the track is longer than a normal drive can reproduce. We can simply remove both of these routines with a small piece of code that returns 0.

If we load up our previously saved loader and then run it with the crack disk in the drive and stop at the insert disk 2 screen we can easily quickly patch out those two routines (the first one starts at \$28736 and if we scroll back through the second one we can see it starts at \$343f2). See figure 8g.

So now we insert disk 2 and continue with the loading. This time we are able to get as far as starting to play the game. So we have clearly gotten further now. However as soon as we start playing the game it crashes within a few seconds of starting. Clearly we have missed something else.

So now we have to start thinking maybe some checksums have been triggered. So if we load up the original game and start making some changes to the loader code and/or unpack code maybe we can make it crash in a similar way. So load up the original and wait for the insert disk 2 screen. Next jump into AR and we can redirect the load and unpack routines into the same area of memory we used at \$280000 and \$290000. Copy the first few instructions of code from the original routines to the new address and then jump back to the original code (see figure 8h and figure 8i) . Then patch the long track checks as we did in 8g.

If we now continue the game as usual, if this is a checksum there's a good chance we will have triggered it. Start the game and you will see that the game does not crash at this point. We now have to figure out what we missed as this problem does not seem to be checksum related.

Let's go back to the very beginning and find out where the problem lies. Read in the boot sector and take a disassembly as we did way back at the start (see figure 1b-1f). Studying the code again, something interesting is highlighted. The code at \$40064 to \$4006c unpacks the code to address \$8 but then starts executing at \$18. If we look at the unpacked code (figures 2a – 2k) there are absolutely no references to the data unpacked to \$8-\$18. If we run our original again and take a note of the values unpacked to \$8-\$18 (see figure 8j) that don't appear to be used we can make a test on our crack to see if this makes any difference.

Load up our previously saved loader again and wait for insert disk 2 then jump into AR. Patch the two long track routines routines as per figure 8g again then put the values into addresses \$8 - \$18 as per figure 8j and then carry on loading our crack to see if this gets us any further (see figure 8k).

Start the game and you will see that appears to have solved our problem. The game now seems to work pretty much correctly. You can now keep playing the game some more and see how we get on. If you head right from the start and keep going right if you survive long enough you will notice that the game eventually crashes out. If we repeat the process we used previously to check for checksums on the original game (figures 8h and 8i). This time we see that the original does crash in the same place as our crack when we start heading right after redirecting the loader, unpacker and taking out the long track checks.

It seems that maybe we have found some checksums after all.

```

~028736 MOVE.L A0,00038706
~02873C BSR 00028754
~028740 BSR 000287A6
~028744 BSR 000287D6
~028748 BSR 00028840
~02874C BSR 000288B2
~028750 BRA 000288FC
;=====
^028754 MOVE.W 00DFF01C,0003870C
~02875E MOVE.W 00DFF01E,0003870E
~028768 MOVE.W 00DFF002,00038710
~028772 MOVE.W 00DFF010,0003870A
~02877C MOVE.W #10,00DFF096
~028784 MOVE.W #2,00DFF09C
~02878C MOVE.W #7FFF,00DFF09E
~028794 MOVE.W #8100,00DFF09E
~02879C ORI.B #78,00BFD100
~0287A4 RTS
;=====
^0287A6 BCLR #7,00BFD100
~0287AE MOVEQ #64,D0
~0287B0 BCLR #3,00BFD100
~0287B8 DBF D0,000287B0
~0287BC BSR 000287C2
~0287C0 RTS

```

Figure 8a – long track check routine (1)

```

~0287C2 MOVE.W #BB8,D6
~0287C6 DBF D6,000287C6
~0287CA BTST #5,00BFE001
~0287D2 BNE 000287CA
~0287D4 RTS
;=====
^0287D6 BTST #4,00BFE001
~0287DE BEQ 000287FC
~0287E0 BSET #1,00BFD100
~0287E8 BCLR #0,00BFD100
~0287F0 BSET #0,00BFD100
~0287F8 BSR 000287C2
~0287FA BRA 000287D6
;=====
^0287FC RTS
;=====
^0287FE BTST #4,00BFE001
~028806 BEQ 0002880C
~028808 BSR 000287D6
~02880A BRA 000287FE
;=====
^02880C MOVE.W #0,D2
~028810 TST.W D2
~028812 BEQ 0002883E
~028814 BCLR #1,00BFD100

```

Figure 8b – long track check routine (2)

```

~02881C NOP
~02881E NOP
~028820 NOP
~028822 BCLR    #0,00BFD100
~02882A NOP
~02882C NOP
~02882E NOP
~028830 BSET    #0,00BFD100
~028838 BSR     000287C2
~02883A SUBQ.W  #1,D2
~02883C BRA     00028810
;=====
^02883E RTS
;=====
^028840 BSR     000287FE
~028842 BCLR    #2,00BFD100
~02884A MOVE.W  #8210,00DFF096
~028852 MOVE.L  00038706,00DFF020
~02885C CLR.W   00DFF024
~028862 BSR     0002889E
~028866 MOVE.W  #4000,00DFF024
~02886E MOVE.W  #B778,00DFF024
~028876 MOVE.W  #B778,00DFF024
~02887E MOVE.L  #30D40,D1
~028884 MOVE.W  00DFF01E,D0

```

Figure 8c – long track check routine (3)

```

~02888A SUBQ.L  #1,D1
~02888C BEQ     00028842
~02888E BTST    #1,D0
~028892 BEQ     00028884
~028894 MOVE.W  #2,00DFF09C
~02889C RTS
;=====
^02889E MOVE.B  00BFDD00,D6
~0288A4 MOVE.B  00BFDD00,D6
~0288AA BTST    #4,D6
~0288AE BEQ     000288A4
~0288B0 RTS
;=====
^0288B2 BSET    #7,00BFD100
~0288BA BSET    #3,00BFD100
~0288C2 BCLR    #3,00BFD100
~0288CA MOVE.W  0003870C,D0
~0288D0 BSET    #F,D0
~0288D4 MOVE.W  D0,00DFF09A
~0288DA MOVE.W  00038710,D0
~0288E0 BSET    #F,D0
~0288E4 MOVE.W  D0,00DFF096
~0288EA MOVE.W  0003870A,D0
~0288F0 BSET    #F,D0
~0288F4 MOVE.W  D0,00DFF09E

```

Figure 8d – long track check routine (4)

```

~0288FA RTS
=====
^0288FC MOVEA.L 00038706,A0
~028902 MOVE.L #3778,D3
~028908 MOVE.L (A0),D0
~02890A ADDQ.L #2,A0
~02890C MOVE.L #F,D2
~028912 MOVE.L D0,D1
~028914 SWAP D1
~028916 CMPI.W #4454,D1
~02891A BEQ 0002892A
~02891C ADD.L D0,D0
~02891E DBF D2,00028912
~028922 DBF D3,00028908
~028926 BRA 0002895A
=====
^02892A MOVEQ #0,D5
~02892C MOVE.L (A0),D0
~02892E ADDQ.L #2,A0
~028930 MOVE.L #F,D2
~028936 MOVE.L D0,D1
~028938 SWAP D1
~02893A CMPI.W #4454,D1
~02893E BEQ 0002894E
~028940 ADD.L D0,D0

```

Figure 8e – long track check routine (5)

```

~028942 DBF D2,00028936
~028946 ADDQ.L #1,D5
~028948 DBF D3,0002892C
~02894C BRA 0002895A
=====
^02894E SUBI.L #1A2C,D5
~028954 BMI 0002895A
~028956 CLR.W D0
~028958 RTS
=====
^02895A MOVE.W #1,D0
~02895E RTS
=====

```

Figure 8f – long track check routine (6)

```

Ready.
In beast load,all, 50000
Loading from 050000 to 051A16
Disk ok
g 50000
No known virus in memory!
Ready.
a 28736
^028736 clr.w d0
^028738 rts
^02873A
d 343f2
~0343F2 MOVE.L A0,00038706
~0343F8 BSR 00034410
~0343FC BSR 00034462

a 343f2
^0343F2 clr.w d0
^0343F4 rts
^0343F6
-

```

Figure 8g – remove the two long track check routines

```

d 38472
^038472 MOVE.L A0,-(A7)
^038474 BSR 0003847C
^038478 MOVEA.L (A7)+,A0
^03847A RTS
=====
^03847C LEA 38308(PC),A3

a 280000
^280000 move.l a0,-(a7)
^280002 jsr 3847c
^280004 move.l (a7)+,a0
^280006 rts
^28000C
a 38472
^038472 jmp 280000
^038478
-

```

Figure 8h – redirect the loader code to \$280000

```

Ready.
d 28960
^028960 ADDA.L (A0),A0
^028962 MOVEA.L -(A0),A2
^028964 ADDA.L A1,A2
^028966 MOVE.L -(A0),D5
a 290000
^290000 adda.l (a0),a0
^290002 move.l -(a0),a2
^290004 add.l a1,a2
^290006 jmp 28966
^29000C
a 28960
^028960 jmp 290000
^028966
-

```

Figure 8i – redirect the unpack code to \$290000

```

Ready.
n 8
:000008 00 05 16 4C 00 05 97 70 00 00 19 0C 00 00 0B 40 ...L...p.....@

```

Figure 8j – data at \$8

```

Ready.
In beast load.all, 50000
Loading from 050000 to 051A16
Disk ok
g 50000
No known virus in memory!
Ready.
d 28960
^028960 JMP      00290000
=====
^028966 MOVE.L  -(A0),D5

a 28736
^028736 clr.w d0
^028738 rts
^02873A
a 343f2
^0343f2 clr.w d0
^0343f4 rts
^0343f6
n 8
:000008 00 05 16 4C 00 05 97 70 00 00 19 0C 00 00 0B 40 ...L...p.....0
:000018 00 FC 08 20 00 FC 08 22 00 FC 09 0E 00 FC 08 26 .ü. .ü. .ü...ü.&
-

```

Figure 8k – remove long track checks and update data at \$8 - \$18

Checksums

Arrggghhhh – checksums – I hate checksums. I spent quite some time trying to figure out the best way to find the checksum. I started off trying to figure out what the trigger was. It doesn't seem to happen at a particular point in the game (although it generally happens roughly in the same place if you follow the same actions). This suggested that it might be time based but if you start the game and don't move from the start position it doesn't crash out

I tried looking through the interrupt code to see if I could find anything suspicious but all to no avail. I also tried a very slow process of trying to track down which area of memory was covered by the checksum using a binary chop method (eg. modify an area of memory see if it crashed, find a mid-point in memory between that test and the previous test and repeat the process again). None of these techniques yielded much further information regarding where the checks were being executed. I even tried searching for quite a few variations on EOR opcodes which are often used for checksum routines.

Eventually I did stumble across the checksum code pretty much completely by accident. I was looking at the low memory addresses looking through the exception vectors and I realised that the value previously put into address \$c (figure 8j) is a valid address and the vector is the address error vector. I decided to investigate the code at \$59770 and found the code in figure 9a. This clearly looks like a checksum routine of some kind. It reads two words from memory and performs an EOR and adds them onto a total. Once the end area of memory (\$3462c) has been reached it checks the result in D6 against \$B6f and jumps off to another piece of code if it's not correct otherwise it resets the address back to \$2e94a and carries on.

However, disabling this checksum (removing the check and branch at \$5979c-597a0) and playing through the game it doesn't appear to help resolve the problem we are seeing, so there must be more than 1 checksum. If there are other checksum routines there is a good chance they may be disguised by using different registers and/or addresses. The BLE instruction struck me as an interesting one since it's relatively uncommonly used instruction and if the routine were similar the same instruction might be used. So I decided to search for opcode \$6f18 and the result was as per figure 9b. Looking at the address found we see the code in figure 9c.

We seem to have found another very similar looking checksum routine (\$29aa0 - \$29ae0). If you disable this checksum by removing the compare and branch at \$29ac8 you will find that the game no longer crashes at the point it did before.

So we have now eliminate two checksums, some more play testing is required so see if we can find any more checksums. Starting the game and heading left everything seems fine. Enter the tree stump when you reach it and you will see the game starts loading but then gets stuck in a loop at that point and does not allow you to continue on with the game.

Jump into AR at this point. As I did so I found myself in the middle of my load routine. Put a breakpoint at the end of the load routine at \$28000e at step through the remainder of the load routine and see where we hit the calling code. You should end up somewhere roughly in the region of the code shown in figure 9d. If you scroll down through the code you will see

the remainder of the routine in figures 9e-9g. Put a breakpoint at the end of the routine (\$3047e) and step to the calling code and you will see you should be at the code shown in figure 9h and 9i. It seems we have found another checksum routine at \$304e0.

Looking at the code we can use a similar approach to the one we used last time searching for the branch opcode to see if we can find any further checks. You should see the results as per figure 9j – two other possible checksum routines. Disassembly of these two routines is shown in figure 9k and 9l.

If you now restart our crack disk, wait for the insert disk 2 prompt and apply all of the changes required to remove all 5 checksums and the 2 long track checks you should be able to fully play the game through to completion (I did so by finding an infinite energy hack and by following a walkthrough on youtube).

```

n c
:00000C 00 05 97 70 00 00 19 0C 00 00 0B 40 72 00 20 38 ...p.....@r. 8
d 59770
~059770 ADDA.W #E,A7
~059774 MOVEA.L 000597BC,A3
~05977A MOVE.W 000597BA,D6
~059780 MOVE.W (A3)+,D1
~059782 MOVE.W (A3)+,D3
~059784 EOR.W D1,D3
~059786 ADD.W D3,D6
~059788 MOVE.W D6,000597BA
~05978E MOVE.L A3,000597BC
~059794 CMPA.L #3462C,A3
~05979A BLE 000597B4
~05979C CMP.W #B6F,D6
~0597A0 BNE 000598E0
~0597A4 CLR.W 000597BA
~0597AA MOVE.L #2E94A,000597BC
~0597B4 JMP 00035506
;=====

```

Figure 9a – a checksum routine 598e0

```

n 5979a
:05979A 6F 18 BC 7C 0B 6F 66 00 01 3E 42 79 00 05 97 BA o...l.of..>By....
f 6f 18, 0 00000
Search from: 000000 to: 000000
029AC6 05979A
Ready.
-

```

Figure 9b –search for opcode \$6f18

```

=====
~029A96 MOVE.W #22F,D1
~029A9A SUB.W D0,D1
~029A9C LSR.W #4,D1
~029A9E ADDQ.W #1,D1
~029AA0 MOVEA.L 0001EA70,A4
~029AA6 MOVE.W (A4)+,D2
~029AA8 MOVE.W (A4)+,D5
~029AAA SUB.W D2,D5
~029AAC MOVE.W 0001EA74,D2
~029AB2 ADD.W D5,D2
~029AB4 MOVE.W D2,0001EA74
~029ABA MOVE.L A4,0001EA70
~029AC0 CMPA.L #348B4,A4
~029AC6 BLE 00029AE0
~029AC8 CMP.W #B40,D2
~029ACC BNE 00029BFA
~029AD0 CLR.W 0001EA74
~029AD6 MOVE.L #31232,0001EA70
~029AE0 MOVEA.W #0,A4
~029AE4 MOVE.W D1,D2
~029AE6 CLR.W D5
~029AE8 MOVE.B D(A6,D4.W),D5
~029AEC MOVE.W D5,D3
~029AEE ADD.W 4(A2),D3

```

Figure 9c – another checksum routine

```

Ready.
Breakpoint raised at address: 0028000E
st
D0=00000000 55500003 000000F5 00000000 00010000 00000000 55555555 00000000
A0=0004A950 000385FE 000121FA 00DFF0A0 000130C0 00BFD500 002804EE 000004F4
PC = 00280010 USP = 00C01496 SR = 201C T=0 S=1 I=000 X=1 N=1 Z=1 V=0 C=0
~280010 CLR.L D0
st
D0=00000000 55500003 000000F5 00000000 00010000 00000000 55555555 00000000
A0=0004A950 000385FE 000121FA 00DFF0A0 000130C0 00BFD500 002804EE 000004F4
PC = 00280012 USP = 00C01496 SR = 2014 T=0 S=1 I=000 X=1 N=0 Z=1 V=0 C=0
~280012 RTS
st
D0=00000000 55500003 000000F5 00000000 00010000 00000000 55555555 00000000
A0=0004A950 000385FE 000121FA 00DFF0A0 000130C0 00BFD500 002804EE 000004F8
PC = 00030334 USP = 00C01496 SR = 2014 T=0 S=1 I=000 X=1 N=0 Z=1 V=0 C=0
~030334 LEA 00046300,A1
~03033A JSR 00028960
~030340 LEA 00050000,A0
~030346 LEA 000385CE,A1
~03034C JSR 00038472
~030352 MOVEA.L A0,A1
~030354 JSR 00028960
~03035A MOVE.L #3462C,0000006C.S
~030362 JSR 0003461C

```

Figure 9d – stuck in a loop

```

~030334 LEA      00046300,A1
~03033A JSR      00028960
~030340 LEA      00050000,A0
~030346 LEA      000385CE,A1
~03034C JSR      00038472
~030352 MOVEA.L  A0,A1
~030354 JSR      00028960
~03035A MOVE.L   #3462C,0000006C.S
~030362 JSR      0003461C
~030368 MOVE.W   #1A0,00DFF096
~030370 MOVE.L   #28448,00DFF080
~03037A MOVE.W   D0,00DFF088
~030380 MOVE.W   #5200,00DFF100
~030388 CLR.W     00DFF102
~03038E CLR.W     00DFF104
~030394 CLR.W     00DFF108
~03039A CLR.W     00DFF10A
~0303A0 MOVE.W   #F4B0,00DFF090
~0303A8 LEA      00DFF180,A0
~0303AE MOVE.W   #1F,D0
~0303B2 CLR.W     (A0)+
~0303B4 DBF      D0,000303B2
~0303B8 LEA      0005BAC0,A0
~0303BE MOVE.W   #254F,D0
~0303C2 CLR.L     (A0)+

```

Figure 9e – stuck in a loop (2)

```

~0303C4 DBF      D0,000303C2
~0303C8 LEA      00046340,A0
~0303CE LEA      0005BDE6,A1
~0303D4 CLR.W     00DFF064
~0303DA MOVE.W   #C,00DFF066
~0303E2 MOVE.W   #FFFF,00DFF044
~0303EA MOVE.W   #FFFF,00DFF046
~0303F2 MOVE.W   #9F0,00DFF040
~0303FA CLR.W     00DFF042
~030400 MOVE.L   A0,00DFF050
~030406 MOVE.W   #4,D0
~03040A MOVE.L   A1,00DFF054
~030410 MOVE.W   #1D4E,00DFF058
~030418 ADDA.L   #1B58,A1
~03041E DBF      D0,0003040A
~030422 LEA      00046300,A0
~030428 LEA      00DFF180,A1
~03042E MOVE.W   #1F,D0
~030432 MOVE.W   (A0)+,(A1)+
~030434 DBF      D0,00030432
~030438 LEA      00046302,A0
~03043E MOVE.W   (A0),00028456
~030444 JSR      0003461C
~03044A MOVE.W   #8180,00DFF096
~030452 LEA      00050000,A0

```

Figure 9f – stuck in a loop(3)

```

~030458 LEA      000648B8,A1
~03045E MOVE.W  #16F,D0
~030462 MOVE.L  (A0)+,(A1)+
~030464 DBF     D0,00030462
~030468 MOVEA.L #4A33C,A0
~03046E MOVEA.L #65000,A1
~030474 MOVE.W  #7C,D0
~030478 MOVE.L  (A0)+,(A1)+
~03047A DBF     D0,00030478
~03047E RTS
=====
~030480 JSR      00000554,$

```

Figure 9g – stuck in a loop(4)

```

Ready.
Breakpoint raised at address: 0003047E
st
D0=0000FFFF 55400003 000000F5 00000000 00010000 00000000 55555555 00000000
A0=0004A530 000651F4 00012EBA 00DFF0A0 000130C0 00BFD500 002804EE 000004FC
PC = 000304A0 USP = 00C01496 SR = 2004 T=0 S=1 I=000 X=0 N=0 Z=1 V=0 C=0
~0304A0 MOVE.L  #65000,00037CE2
~0304AA MOVE.L  00037CE2,00037CEE
~0304B4 MOVE.W  #1,00037CE6
~0304BC MOVE.W  #1,00037CE8
~0304C4 LEA     00010000,A0
~0304CA BSR     00028736
~0304CE TST.W   D0
~0304D0 BNE     00035036
~0304D4 JSR     0003830C
~0304DA JSR     000383DE
~0304E0 LEA     000284D4,A0
~0304E6 LEA     00029E0C,A1
~0304EC MOVEQ   #0,D2
~0304EE MOVE.W  (A0)+,D0
~0304F0 MOVE.W  (A0)+,D1
~0304F2 EOR.W   D0,D1
~0304F4 ADD.W   D1,D2
~0304F6 CMPA.L  A0,A1
~0304F8 BGT     000304EE

```

Figure 9h – possible checksum (1)

```

~0304FA CMP.W   #5939,D2
~0304FE BNE     000306F4
~030502 MOVE.W  #1,000382DA
~03050A JSR     00028AFC
~030510 LEA     00044E00,A0
~030516 LEA     0003860E,A1
~03051C JSR     00038472
~030522 MOVEA.L A0,A1
~030524 JSR     00028960
~03052A LEA     00046300,A0
~030530 LEA     00038616,A1
~030536 JSR     00038472
~03053C MOVEA.L A0,A1
~03053E JSR     00028960
~030544 MOVE.L  #67FA0,00037B04
~03054E BSR     00030FEA
~030552 MOVE.L  #67FA0,000382FA
~03055C LEA     00000500,$,A0
~030560 LEA     00038626,A1
~030566 JSR     00038472
~03056C JSR     00038412
~030572 TST.W   00037B6E
~030578 BEQ     00030572
~03057A CLR.W   00037CE8
~030580 CLR.W   00037CEA

```

Figure 9i – possible checksum (2)

```
n 304f8
:0304F8 6E F4 B4 7C 59 39 66 00 01 F4 33 FC 00 01 00 03 n..lY9f...3ü...
f 6e f4, 0 80000
Search from: 000000 to: 000000
01ED18 0304F8 0312FE
Ready.
```

Figure 9j – look for other similar routines

```
d 1ed00
~01ED00 LEA    0003018C,A1
~01ED06 LEA    00031A86,A2
~01ED0C MOVEQ  #0,D5
~01ED0E MOVE.W (A1)+,D4
~01ED10 MOVE.W (A1)+,D6
~01ED12 SUB.W  D4,D6
~01ED14 ADD.W  D6,D5
~01ED16 CMPA.L A1,A2
~01ED18 BGT    0001ED0E
~01ED1A CMP.W  #BCC5,D5
~01ED1E BNE    0001EBA4
~01ED22 JSR    0003461C
~01ED28 MOVE.W #6600,00DFF100
```

Figure 9k – checksum number 4

```
d 312e0
~0312E0 JSR    00038472
~0312E6 LEA    0003264C,A0
~0312EC LEA    000351BE,A1
~0312F2 MOVEQ  #0,D2
~0312F4 MOVE.W (A0)+,D0
~0312F6 MOVE.W (A0)+,D1
~0312F8 EOR.W  D0,D1
~0312FA ADD.W  D1,D2
~0312FC CMPA.L A0,A1
~0312FE BGT    000312F4
~031300 CMP.W  #3073,D2
~031304 BNE    0002E938
~031308 BSR    00038412
~03130C BSR    0003137C
```

Figure 9l –checksum number 5

Finally putting it all together

Our last steps will be to combine all the stuff we found into our loader, find a proper area of memory where we can locate our load and unpack code (remember we placed it in fast ram temporarily) and then make our loader into an executable and create a bootable disk.

Firstly then lets add the long track routine and the checksum stuff to our loader. See figure 10a and 10b then update the copy code at the start of our loader to make sure it reflects the length of the patching code correctly and save the loader.

Run the game though with our new loader and make sure everything looks good.

Next job is to find some spare memory for our routines. The space used by the routines is around \$20c (524) bytes for the unpack routine and \$514 (1300) bytes for the disk load routines. The existing load and unpack routines are very compact and we can't simply replace them with our new routines. For this reason we need to try and find some spare memory.

Simply fill all the memory in AR with a chosen value then load up our loader and start the game (see figure 10c). Play through the game and then jump back into AR to see if we can find some memory to place our stuff (see figure 10d – 10g).

From these results it seems that there is some unused memory from \$84 - \$1d4 and some also \$5b4d6 - \$5ba88. That's not a lot of unused memory – around \$5b2 (1458) bytes at \$5b4d6 and \$150 (336) bytes at \$84. There's enough memory at \$5b4d6 to fit our load routine but not the unpack routine as well.

I've used the RNC unpack routines quite a few times now and I know that they are very stack hungry. In the past I've had to force them to use other areas of memory as they used too much stack space. We might have more usable low memory if the RNC unpack routine is forced to use a different area of memory.(see figure 10h and 10i).

Run the game through again and see how our spare memory compares again. See the results in figure 10j and 10k. Now we have around \$2d0 (720) bytes of low memory – plenty of room for our unpack code. Now we need to find temporary area for our unpack code. I've had success in the past using the loader MFM buffer as a temporary workspace during unpack so I think that's a good thing to try here.

We just need to find some spare memory to put our patch routine now so we repeat the same process and check the memory usage at the first insert disk 2 prompt. We can see unused memory at \$60000 in figure 10l.

Ok so we put all this together and our final code is shown in figures 10m through 10s. Firstly we copy the original loader to \$68000 (and we have made some changes in this routine to jump to our new load routine at \$5b500, our depack code at \$120 and our patcher code at \$60000). Then we copy our loader code to \$5b500. Next we copy our unpack code to to address \$120 and our patcher code to \$60000 then we start executing the original loader using a trap #0. The changes to pick up the MFM buffer address have been included into our

unpack code at \$517b6 and the final patch code can be seen at \$519be. The final loading code is included in the archive as BEAST_LOAD_FINAL

We now need to convert this binary file into an executable and run it from the crack disk 1.

I simply wrote the following code in devpac

```
INCBIN "beast_load_final"
```

and assembled it to produce an exe. Copy the exe to the disk 1 and create a startup-sequence and install the disk and you should be good to go. The final output exe is included in the archive as BEAST_LOAD.EXE

```
d 51a10
~051A10 LEA    00028736,A0
~051A16 MOVE.L #42404E75,D0
~051A1C MOVE.L D0,(A0)
~051A1E LEA    000343F2,A0
~051A24 MOVE.L D0,(A0)
~051A26 LEA    0005979C,A0
~051A2C MOVE.L #4E714E71,D0
~051A32 MOVE.W #3C3C,(A0)
~051A36 MOVE.L D0,4(A0)
~051A3A LEA    00029AC8,A0
~051A40 MOVE.W #343C,(A0)
~051A44 MOVE.L D0,4(A0)
~051A48 LEA    000304FA,A0
~051A4E MOVE.W #343C,(A0)
~051A52 MOVE.L D0,4(A0)
~051A56 LEA    0001ED1A,A0
~051A5C MOVE.W #3A3C,(A0)
~051A60 MOVE.L D0,4(A0)
~051A64 LEA    00031300,A0
~051A6A MOVE.W #343C,(A0)
~051A6E MOVE.L D0,4(A0)
~051A72 LEA    00000000,A0
~051A78 MOVE.L #5164C,(A0)+
~051A7E MOVE.L #59770,(A0)+
```

Figure 10a – append the necessary code onto the end of our loader (1)

```
~051A84 MOVE.L #190C,(A0)+
~051A8A MOVE.L #B40,(A0)
~051A90 JMP    0001E986
=====
~051A96 ORI.B  #0,D0
```

Figure 10a – append the necessary code onto the end of our loader (2)

```

d 50036
~050036 MOVE.W (A0)+,(A2)+
~050038 CMPA.L A0,A1
~05003A BNE 00050036
~05003C LEA 519BE(PC),A0
~050040 LEA 51A96(PC),A1
~050044 LEA 002A0000,A2
~05004A MOVE.L (A0)+,(A2)+

sm beast_load.all, 50000 51a96
Disk ok

```

Figure 10b – update loader to copy all the new patching code to \$2a0000

```

(p) by Date L
*****
No known virus in memory!
Ready.
o 55, 80 7ffff
Ready.
lm beast_load.all, 50000
Loading from 050000 to 051A96
Disk ok
g 50000
No known virus in memory!
Ready.

```

Figure 10c – fill the memory with fixed value

```

f 55 55 55 55 55 55 55 55, 0 80000
Search from: 000000 to: 000000
000004 000005 000006 000007 000008 000009 00000A 00000B 00000C 00000D
00000E 00000F 000010 000011 000012 000013 000014 000015 000016 000017
000018 000019 00001A 00001B 00001C 00001D 00001E 00001F 000020 000021
000022 000023 000024 000025 000026 000027 000028 000029 00002A 00002B
00002C 00002D 00002E 00002F 000030 000031 000032 000033 000034 000035
000036 000037 000038 000039 00003A 00003B 00003C 00003D 00003E 00003F
000040 000041 000042 000043 000044 000045 000046 000047 000048 000049
00004A 00004B 00004C 00004D 00004E 00004F 000050 000051 000052 000053
000054 000055 000056 000057 000058 000059 00005A 00005B 00005C 00005D
00005E 00005F 000060 000061 000062 000063 000064 000065 000066 000067
000068 000069 00006A 00006B 00006C 00006D 00006E 00006F 000070 000071
000072 000073 000074 000075 000076 000077 000078 000079 00007A 00007B
00007C 00007D 00007E 00007F 000080 000081 000082 000083 000084 000085
000086 000087 000088 000089 00008A 00008B 00008C 00008D 00008E 00008F
000090 000091 000092 000093 000094 000095 000096 000097 000098 000099
00009A 00009B 00009C 00009D 00009E 00009F 0000A0 0000A1 0000A2 0000A3
0000A4 0000A5 0000A6 0000A7 0000A8 0000A9 0000AA 0000AB 0000AC 0000AD
0000AE 0000AF 0000B0 0000B1 0000B2 0000B3 0000B4 0000B5 0000B6 0000B7
0000B8 0000B9 0000BA 0000BB 0000BC 0000BD 0000BE 0000BF 0000C0 0000C1
0000C2 0000C3 0000C4 0000C5 0000C6 0000C7 0000C8 0000C9 0000CA 0000CB
0000CC 0000CD 0000CE 0000CF 0000D0 0000D1 0000D2 0000D3 0000D4 0000D5
0000D6 0000D7 0000D8 0000D9 0000DA 0000DB 0000DC 0000DD 0000DE 0000DF
0000E0 0000E1 0000E2 0000E3 0000E4 0000E5 0000E6 0000E7 0000E8 0000E9
0000EA 0000EB 0000EC 0000ED 0000EE 0000EF 0000F0 0000F1 0000F2 0000F3
0000F4 0000F5 0000F6 0000F7 0000F8 0000F9 0000FA 0000FB 0000FC 0000FD
0000FE 0000FF 000100 000101 000102 000103 000104 000105 000106 000107
000108 000109 00010A 00010B 00010C 00010D 00010E 00010F 000110 000111
000112 000113 000114 000115 000116 000117 000118 000119 00011A 00011B
00011C 00011D 00011E 00011F 000120 000121 000122 000123 000124 000125
000126 000127 000128 000129 00012A 00012B 00012C 00012D 00012E 00012F
000130 000131 000132 000133 000134 000135 000136 000137 000138 000139
00013A 00013B 00013C 00013D 00013E 00013F 000140 000141 000142 000143
000144 000145 000146 000147 000148 000149
Ready.

```

Figure 10d – find spare memory (1)

```

000106 000107 000108 000109 00010A 00010B 00010C 00010D 00010E 00010F
000110 000111 000112 000113 000114 000115 000116 000117 000118 000119
00011A 00011B 00011C 00011D 00011E 00011F 000120 000121 000122 000123
000124 000125 000126 000127 000128 000129 00012A 00012B 00012C 00012D
00012E 00012F 000130 000131 000132 000133 000134 000135 000136 000137
000138 000139 00013A 00013B 00013C 00013D 00013E 00013F 000140 000141
000142 000143 000144 000145 000146 000147 000148 000149 00014A 00014B
00014C 00014D 00014E 00014F 000150 000151 000152 000153 000154 000155
000156 000157 000158 000159 00015A 00015B 00015C 00015D 00015E 00015F
000160 000161 000162 000163 000164 000165 000166 000167 000168 000169
00016A 00016B 00016C 00016D 00016E 00016F 000170 000171 000172 000173
000174 000175 000176 000177 000178 000179 00017A 00017B 00017C 00017D
00017E 00017F 000180 000181 000182 000183 000184 000185 000186 000187
000188 000189 00018A 00018B 00018C 00018D 00018E 00018F 000190 000191
000192 000193 000194 000195 000196 000197 000198 000199 00019A 00019B
00019C 00019D 00019E 00019F 0001A0 0001A1 0001A2 0001A3 0001A4 0001A5
0001A6 0001A7 0001A8 0001A9 0001AA 0001AB 0001AC 0001AD 0001AE 0001AF
0001B0 0001B1 0001B2 0001B3 0001B4 0001B5 0001B6 0001B7 0001B8 0001B9
0001BA 0001BB 0001BC 0001BD 0001BE 0001BF 0001C0 0001C1 0001C2 0001C3
0001C4 0001C5 0001C6 0001C7 0001C8 0001C9 0001CA 0001CB 0001CC 0001CD
0001CE 0001CF 0001D0 0001D1 0001D2 0001D3 0001D4 000268 000269 00026A
00026B 00026C 00026D 00026E 00026F 000270 000271 000272 000273 000274
000275 000276 000277 000278 0002A8 0002A9 0002AA 0002AB 0002AC 0002AD
0002AE 0002AF 0002B0 000368 000369 00036A 00036B 00036C 00036D 00036E
00036F 000370 018508 _

```

Figure 10e - find spare memory (2)

```

00017E 00017F 000180 000181 000182 000183 000184 000185 000186 000187
000188 000189 00018A 00018B 00018C 00018D 00018E 00018F 000190 000191
000192 000193 000194 000195 000196 000197 000198 000199 00019A 00019B
00019C 00019D 00019E 00019F 0001A0 0001A1 0001A2 0001A3 0001A4 0001A5
0001A6 0001A7 0001A8 0001A9 0001AA 0001AB 0001AC 0001AD 0001AE 0001AF
0001B0 0001B1 0001B2 0001B3 0001B4 0001B5 0001B6 0001B7 0001B8 0001B9
0001BA 0001BB 0001BC 0001BD 0001BE 0001BF 0001C0 0001C1 0001C2 0001C3
0001C4 0001C5 0001C6 0001C7 0001C8 0001C9 0001CA 0001CB 0001CC 0001CD
0001CE 0001CF 0001D0 0001D1 0001D2 0001D3 0001D4 000268 000269 00026A
00026B 00026C 00026D 00026E 00026F 000270 000271 000272 000273 000274
000275 000276 000277 000278 0002A8 0002A9 0002AA 0002AB 0002AC 0002AD
0002AE 0002AF 0002B0 000368 000369 00036A 00036B 00036C 00036D 00036E
00036F 000370 018508 05B4D6 05B4D7 05B4D8 05B4D9 05B4DA 05B4DB 05B4DC
05B4DD 05B4DE 05B4DF 05B4E0 05B4E1 05B4E2 05B4E3 05B4E4 05B4E5 05B4E6
05B4E7 05B4E8 05B4E9 05B4EA 05B4EB 05B4EC 05B4ED 05B4EE 05B4EF 05B4F0
05B4F1 05B4F2 05B4F3 05B4F4 05B4F5 05B4F6 05B4F7 05B4F8 05B4F9 05B4FA
05B4FB 05B4FC 05B4FD 05B4FE 05B4FF 05B500 05B501 05B502 05B503 05B504
05B505 05B506 05B507 05B508 05B509 05B50A 05B50B 05B50C 05B50D 05B50E
05B50F 05B510 05B511 05B512 05B513 05B514 05B515 05B516 05B517 05B518
05B519 05B51A 05B51B 05B51C 05B51D 05B51E 05B51F 05B520 05B521 05B522
05B523 05B524 05B525 05B526 05B527 05B528 05B529 05B52A 05B52B 05B52C
05B52D 05B52E 05B52F 05B530 05B531 05B532 05B533 05B534 05B535 05B536
05B537 05B538 05B539 05B53A 05B53B 05B53C 05B53D 05B53E 05B53F 05B540
05B541 05B542 05B543 05B544 05B545 05B546 05B547 05B548 05B549 05B54A
05B54B 05B54C 05B54D 05B54E 05B54F 0._

```

Figure 10f - find spare memory (3)

```

05B9D3 05B9D4 05B9D5 05B9D6 05B9D7 05B9D8 05B9D9 05B9DA 05B9DB 05B9DC
05B9DD 05B9DE 05B9DF 05B9E0 05B9E1 05B9E2 05B9E3 05B9E4 05B9E5 05B9E6
05B9E7 05B9E8 05B9E9 05B9EA 05B9EB 05B9EC 05B9ED 05B9EE 05B9EF 05B9F0
05B9F1 05B9F2 05B9F3 05B9F4 05B9F5 05B9F6 05B9F7 05B9F8 05B9F9 05B9FA
05B9FB 05B9FC 05B9FD 05B9FE 05B9FF 05BA00 05BA01 05BA02 05BA03 05BA04
05BA05 05BA06 05BA07 05BA08 05BA09 05BA0A 05BA0B 05BA0C 05BA0D 05BA0E
05BA0F 05BA10 05BA11 05BA12 05BA13 05BA14 05BA15 05BA16 05BA17 05BA18
05BA19 05BA1A 05BA1B 05BA1C 05BA1D 05BA1E 05BA1F 05BA20 05BA21 05BA22
05BA23 05BA24 05BA25 05BA26 05BA27 05BA28 05BA29 05BA2A 05BA2B 05BA2C
05BA2D 05BA2E 05BA2F 05BA30 05BA31 05BA32 05BA33 05BA34 05BA35 05BA36
05BA37 05BA38 05BA39 05BA3A 05BA3B 05BA3C 05BA3D 05BA3E 05BA3F 05BA40
05BA41 05BA42 05BA43 05BA44 05BA45 05BA46 05BA47 05BA48 05BA49 05BA4A
05BA4B 05BA4C 05BA4D 05BA4E 05BA4F 05BA50 05BA51 05BA52 05BA53 05BA54
05BA55 05BA56 05BA57 05BA58 05BA59 05BA5A 05BA5B 05BA5C 05BA5D 05BA5E
05BA5F 05BA60 05BA61 05BA62 05BA63 05BA64 05BA65 05BA66 05BA67 05BA68
05BA69 05BA6A 05BA6B 05BA6C 05BA6D 05BA6E 05BA6F 05BA70 05BA71 05BA72
05BA73 05BA74 05BA75 05BA76 05BA77 05BA78 05BA79 05BA7A 05BA7B 05BA7C
05BA7D 05BA7E 05BA7F 05BA80 05BA81 05BA82 05BA83 05BA84 05BA85 05BA86
05BA87 05BA88 05BA89 05BA8A 05BA8B 05BA8C 05BA8D 05BA8E 05BA8F 05BA90
05BA91 05BA92 05BA93 05BA94 05BA95 05BA96 05BA97 05BA98 05BA99 05BA9A
05BA9B 05BA9C 05BA9D 05BA9E 05BA9F 05BAA0 05BAA1 05BAA2 05BAA3 05BAA4
05BAA5 05BAA6 05BAA7 05BAA8 05BAA9 05BAAA 05BAAB 05BAAC 05BAAD 05BAAE
05BAAF 05BAB0 05BAB1 05BAB2 05BAB3 05BAB4 05BAB5 05BAB6 05BAB7 05BAB8
Ready.

```

Figure 10g - find spare memory (4)

```

No known virus in memory!
Ready.
o 55, 80 7ffff
Ready.
In beast load.all, 50000
Loading from 050000 to 051A96
Disk ok
d 517b2
~0517B2 MOVEM.L D0-D7/A0-A6,-(A7)
~0517B6 LEA -180(A7),A7
~0517BA MOVEA.L A7,A2
~0517BC BSR 0005192A
a 517b6
^0517B6 lea 53000(pc),a2
^0517BA nop
^0517BC
d 517cc
~0517CC BSR 0005192A
~0517D0 MOVE.L D0,180(A7)
~0517D4 LEA A(A0),A3
a 517d0
^0517D0 move.l d0,(a7)
^0517D2 nop
^0517D4

```

Figure 10h – update RNC unpack to reduce stack usage (1)

```

d 518c2
~0518C2 MOVE.L D1,180(A7)
~0518C6 LEA 180(A7),A7
~0518CA MOVEM.L (A7)+,D0-D7/A0-A6
~0518CE RTS
;=====
^0518D0 MOVE.W (A0)+,D0
a 518c2
^0518C2 move.l d1,(a7)
^0518C4 nop
^0518C6 nop
^0518C8 nop
^0518CA

```

[illegible][illegible]

```

^0518C4 nop
^0518C6 nop
^0518C8 nop
^0518CA
g 50000
No known virus in memory!
Ready.
h 60000
:060000 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060010 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060020 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060030 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060040 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060050 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060060 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060070 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060080 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060090 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:0600A0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:0600B0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:0600C0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:0600D0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:0600E0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:0600F0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU
:060100 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 UUUUUUUUUUUUUUUUUUUUU

```

Figure 10l – spare memory for patch routine

```

d 50000
~050000 LEA 5005E(PC),A0
~050004 LEA 5129E(PC),A1
~050008 LEA 00068000,A2
~05000E MOVE.W (A0)+,(A2)+
~050010 CMPA.L A0,A1
~050012 BNE 0005000E
~050014 LEA 5129E(PC),A0
~050018 LEA 517B2(PC),A1
~05001C LEA 0005B500,A2
~050022 MOVE.W (A0)+,(A2)+
~050024 CMPA.L A0,A1
~050026 BNE 00050022
~050028 LEA 517B2(PC),A0
~05002C LEA 519BE(PC),A1
~050030 LEA 00000120,A2
~050036 MOVE.W (A0)+,(A2)+
~050038 CMPA.L A0,A1
~05003A BNE 00050036
~05003C LEA 519BE(PC),A0
~050040 LEA 51A96(PC),A1
~050044 LEA 00060000,A2
~05004A MOVE.W (A0)+,(A2)+
~05004C CMPA.L A0,A1

```

Figure 10m – final loader code (1)

```

~05004E BNE 0005004A
~050050 LEA 00068050,A0
~050056 MOVE.L A0,00000080
~05005C TRAP #0
~05005E ORI 2 #200F2 -(A0)

```

Figure 10n – final loader code (2)

```

~05129E MOVEM.L A0-A1,-(A7)
~0512A2 BSR      000512BA
~0512A4 MOVEM.L (A7)+,A0-A1
~0512A8 BTST     #7,(A1)
~0512AC BNE      000512B2
~0512AE CLR.L    D0
~0512B0 RTS
;=====
~0512B2 MOVEA.L  A0,A1
~0512B4 JMP      00000120
;=====
~0512BA MOVE.L   (A1),D0
~0512BC MOVEA.L 000691F8,A3
~0512C2 LEA      2EC0(A3),A2
~0512C6 LEA      51302(PC),A1
~0512CA LEA      512EC(PC),A4
~0512CE MOVEQ    #5,D6
~0512D0 MOVE.B   D0,D7
~0512D2 ANDI.W   #F,D7
~0512D6 MOVE.B   0(A4,D7.W),D7
~0512DA MOVE.B   D7,0(A1,D6.W)
~0512DE LSR.L    #4,D0
~0512E0 DBF      D6,000512D0
~0512E4 LEA      512FC(PC),A1
~0512E8 BSR      0005130A

```

Figure 10o – final loader code (3)

```

d 517b2
~0517B2 MOVEM.L D0-D7/A0-A6,-(A7)
~0517B6 MOVEA.L 0005B520,A2
~0517BC MOVEA.L (A2),A2
~0517BE BSR      0005192A
~0517C2 MOVEQ    #0,D1
~0517C4 CMP.L    #524E4301,D0
~0517CA BNE      000518C2
~0517CE BSR      0005192A
~0517D2 MOVE.L   D0,(A7)
~0517D4 LEA      A(A0),A3
~0517D8 MOVEA.L  A1,A5
~0517DA LEA      0(A5,D0.L),A6
~0517DE BSR      0005192A
~0517E2 LEA      0(A3,D0.L),A4
~0517E6 CLR.W    -(A7)
~0517E8 CMPA.L   A4,A5
~0517EA BCC      00051838
~0517EC MOVEQ    #0,D0
~0517EE MOVE.B   -2(A3),D0
~0517F2 LEA      0(A6,D0.L),A0
~0517F6 CMPA.L   A4,A0
~0517F8 BLS      00051838
~0517FA ADDQ.W   #2,A7
~0517FC MOVE.L   A4,D0

```

Figure 10p – final loader code (4)

```

~0518BA MOVE.B D1,(A5)+
~0518BC SUBQ.B #1,D0
~0518BE BNE 000518B8
~0518C0 BRA 000518C6
=====
~0518C2 MOVE.L D1,(A7)
~0518C4 NOP
~0518C6 NOP
~0518C8 NOP
~0518CA MOVEM.L (A7)+,D0-D7/A0-A6
~0518CE RTS
=====
~0518D0 MOVE.W (A0)+,D0
~0518D2 AND.W D6,D0
~0518D4 SUB.W (A0)+,D0
~0518D6 BNE 000518D0
~0518D8 MOVE.B 3C(A0),D1
~0518DC SUB.B D1,D7
~0518DE BGE 000518E2
~0518E0 BSR 00051912
~0518E2 LSR.L D1,D6
~0518E4 MOVE.B 3D(A0),D0
~0518E8 CMP.B #2,D0
~0518EC BLT 00051904
~0518EE SUBQ.B #1,D0

```

Figure 10q – final loader code (5)

```

~0519B4 MOVEM.L (A7)+,D5-D7
~0519B8 LEA 10(A7),A7
~0519BC RTS
=====
~0519BE LEA 00038472,A0
~0519C4 MOVE.W #4EF9,(A0)+
~0519C8 MOVE.L #5B500,(A0)+
~0519CE LEA 00028960,A0
~0519D4 MOVE.W #4EF9,(A0)+
~0519D8 MOVE.L #120,(A0)+
~0519DE LEA 000385A6,A0
~0519E4 BSET #7,20(A0)
~0519EA BSET #7,80(A0)
~0519F0 BSET #7,E8(A0)
~0519F6 BSET #7,108(A0)
~0519FC BSET #7,148(A0)
~051A02 LEA 0005B51E,A0
~051A08 MOVE.L #382FA,2(A0)
~051A10 LEA 00028736,A0
~051A16 MOVE.L #42404E75,D0
~051A1C MOVE.L D0,(A0)
~051A1E LEA 000343F2,A0
~051A24 MOVE.L D0,(A0)
~051A26 LEA 0005979C,A0
~051A2C MOVE.L #4E714E71,D0

```

Figure 10r – final loader code (6)

```

~051A32 MOVE.W #3C3C,(A0)
~051A36 MOVE.L D0,4(A0)
~051A3A LEA 00029AC8,A0
~051A40 MOVE.W #343C,(A0)
~051A44 MOVE.L D0,4(A0)
~051A48 LEA 000304FA,A0
~051A4E MOVE.W #343C,(A0)
~051A52 MOVE.L D0,4(A0)
~051A56 LEA 0001ED1A,A0
~051A5C MOVE.W #3A3C,(A0)
~051A60 MOVE.L D0,4(A0)
~051A64 LEA 00031300,A0
~051A6A MOVE.W #343C,(A0)
~051A6E MOVE.L D0,4(A0)
~051A72 LEA 00000008,A0
~051A78 MOVE.L #5164C,(A0)+
~051A7E MOVE.L #59770,(A0)+
~051A84 MOVE.L #190C,(A0)+
~051A8A MOVE.L #B40,(A0)
~051A90 JMP 0001E986
=====

```

Figure 10s – final loader code (6)

Playing through the game

Now that we have a fully working crack disk we can play through the game as much as possible and give it a thorough playtesting.

I managed to complete the game without any issues by giving myself infinite energy.

However during further play testing I managed to find one final issue. If you start the game and proceed left to into the castle it does the loading and then asks you to insert disk 1 prior to starting the castle level. If you then die while disk 1 is inserted it attempts to load 2 files from disk 1 that we previously assumed were on disk 2. If your loader has error checking to ensure the files being loaded are present you will easily spot this problem and you can breakpoint the loader to see the two files that it is trying to load that don't exist.

Beast_0e25f0

Beast_0e63f0

These are the last two files on disk 2. It seems that these two files are duplicated on both disks. Simply copy the files over from disk 2 to disk 1 and all will be good.

All done – the end

Everything is taken care of and we now have a fully working cracked copy of Shadow of the Beast.

Thanks to the packing tools that we have available these days we have managed to retain all the data from both disks and keep the disk swapping down to a minimum (no changes from the original).



See you next time...